

Adam Doligalski

Kurs asemblera dla początkujących



Amiga 500-4000



Helion

Adam Doligalski

Amiga 500 – 4000
Kurs asemblera dla początkujących



Gdy minie pierwsza fascynacja komputerem, przychodzi pora na zastanowienie do czego można go wykorzystać... Czy gry i programy użytkowe to wszystko? Czy można samodzielnie programować komputer?

Do tworzenia programów przeznaczone są języki programowania. Jednak wykorzystanie ich do tworzenia animacji może zakończyć się niepowodzeniem, gdyż obraz nie będzie się zmieniał płynnie lecz skokowo. Żeby zwiększyć szybkość działania trzeba tworzyć procedury w assemblerze — języku zrozumiałym dla procesora.

Książka „Amiga 500 – 4000. Kurs assemblera dla początkujących” umożliwi bezpośrednie programowanie procesora osobom, które chcą tworzyć szybkie i efektywne programy, a dotychczas nie zetknęły się z takimi pojęciami jak rejestr czy lista rozkazów procesora.

Projekt okładki: Artgraf

Autor oraz Wydawnictwo Helion dołożyli wszelkich starań, by zawarte w książce informacje były kompletne i rzetelne, nie bierze jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Występujące w tekście znaki są zastrzeżonymi znakami firmowymi bądź towarowymi ich posiadaczy.

ISBN 83-85701-37-0

© HELION, 1994

Printed in Poland

Spis treści

Rozdział I

Wprowadzenie	9
1.1 Systemy liczenia	9
1.2 Bity, bajty, słowa	10
1.3 Organizacja pamięci	10
1.4 Co to jest asembler i kod maszynowy	11

Rozdział II

Mikroprocesor MC 68000	12
2.1 Mikroprocesor Motorola 68000	12
2.2 Rozkazy Motoroli 68000	14
2.3 Tryby adresowania	15
2.4 Kod BCD	19
2.5 Błędne rozkazy a program asemblerujący	19
2.6 Pisanie procedur wykonujących operacje arytmetyczne	20

Rozdział III

Hardware Amigi	33
3.1 Rodzaje pamięci w Amigach	33
3.2 Obszary pamięci i ich znaczenie	33
3.3 Procesory specjalizowane Amigi	34
3.4 Kanały DMA	35

Rozdział IV

Programowanie coppera	36
4.1 Charakterystyka coppera	36
4.2 MOVE, WAIT, SKIP — instrukcje coppera	37
4.2.1 Jak w asemblerze zapisać instrukcję coppera?	37
4.2.2 Instrukcja WAIT	38
4.2.3 Instrukcja MOVE	39
4.2.4 Instrukcja SKIP	39
4.3 Pozioma pozycja promienia wizji w instrukcjach coppera	40
4.4 Pionowa pozycja promienia wizji w instrukcjach coppera	40
4.5 Rejestry coppera	40
4.5.1 Rejestry lokacji	40
4.5.2 Rejestry skoków	41
4.5.3 Rejestr kontrolny	41
4.6 Rejestry kolorów	41
4.7 Tworzenie copperlisty	42

4.8	Kompletne procedury uruchamiające copperlistę	42
4.9	Animacja	49
Rozdział V		
Playfield		56
5.1	Wstęp	56
5.2	Podstawowe cechy playfieldu	56
5.3	Zasady tworzenia prostego playfieldu	57
5.3.1	Podstawowe rejestry — bitplany i kolory	57
5.3.2	Przydzielanie pamięci na bitplany	59
5.3.3	Definiowanie rozmiaru okna wyświetlania	60
5.3.3.1	Ustawianie pozycji startowej i końcowej pionu.	60
5.3.3.2	Ustawianie pozycji startowej i końcowej poziomu.	61
5.3.3.3	Zapisywanie rejestrów.	61
5.3.4	Sposób pobierania danych przez system	62
5.3.5	Rozdzielczości	63
5.3.6	Określenie zawartości bitplanów	64
5.3.7	Wielokrotne wyświetlanie playfieldu	64
5.3.8	Podsumowanie wiadomości o wyświetlaniu playfieldu	64
5.3.9	Przykłady tworzenia prostego playfieldu	65
5.3.10	Tryb interlace	69
5.3.11	Rawblit	71
5.3.12	Tryb EHB (Extra Half Bright)	73
5.3.13	Tryb HAM (Hold And Modify)	73
5.4	Wyświetlanie w trybie DUAL PLAYFIELD	74
5.4.1	Wyznaczanie bitplanów w trybie DUAL PLAYFIELD	74
5.4.2	Rejestry kolorów w trybie DUAL PLAYFIELD	75
5.4.3	Priorytety playfieldów	75
5.4.4	Uruchomienie trybu DUAL PLAYFIELD	75
5.4.5	Podsumowanie wiadomości o trybie DUAL PLAYFIELD	76
5.5	Bitplany i okna wyświetlania w wszystkich rozmiarów	76
5.5.1	Bitplany większe od okna wyświetlania.	76
5.5.2	Maksymalny rozmiar okna wyświetlania	77
5.6	Scrollowanie playfieldów	77
5.6.1	Scrollowanie poziome	78
5.6.1.1	Określanie pobierania danych w scrollowaniu poziomym	78
5.6.1.2	Określanie modulo w scrollowaniu poziomym	78
5.6.1.3	Określanie przesunięcia	79
5.6.1.4	Dokładne omówienie zasady scrollowania	79
5.6.1.5	Przykład scrollowania poziomego	79
5.6.2	Scrollowanie pionowe	92
5.6.2.1	Przykład scrollowania pionowego	92
Rozdział VI		
Blitter		93
6.1	Wprowadzenie	93
6.2	Określanie adresów kanałów i typu operacji.	93
6.3	Określanie obszaru operacji.	97
6.4	Przesunięcia i maski.	98

6.5	Tryb DESCENDING	98
6.6	Tryb wypełniania	99
6.7	Używanie wyłączonych kanałów źródłowych	100
6.8	Bit wykonania i znacznik przerwania	100
6.9	Znacznik zerowy	100
6.10	Rysowanie linii	101
6.10.1	Podsumowanie wiadomości o rysowaniu linii	103
6.11	Reguła programowania blittera	104
6.12	Kontrolowanie blittera za pomocą coppera.	104
6.13	Przykłady programowania blittera.	105
Rozdział VII		
Spritey		109
7.1	Wprowadzenie	109
7.2	Definiowanie struktury spritea dla kanałów DMA.	109
7.2.1	Definiowanie pozycji	110
7.2.1.1	Definiowanie pozycji poziomej	110
7.2.1.2	Definiowanie pozycji pionowej	110
7.2.2	Definiowanie rozmiaru spriteów	111
7.2.3	Kształt spriteów	111
7.2.4	Kolory spriteów	111
7.2.5	Tworzenie struktury spritea	112
7.2.5.1	Pierwsze słowo struktury	113
7.2.5.2	Drugie słowo struktury	113
7.2.5.3	Zapis koloru i kształtu spritea	113
7.2.5.4	Ponowne użycie spritea	114
7.2.5.5	Znacznik końca struktury spritea	114
7.3	Wyświetlanie spriteów	114
7.3.1	Wybieranie kanału DMA i określanie adresów	114
7.3.2	Odświeżanie adresu struktury	115
7.4	Priorytety spriteów	115
7.5	Łączenie spriteów	115
7.5.1	Spritey 16 kolorowe	116
7.6	Tryb ręczny obsługi spriteów	117
7.7	Kolizje między spriteami, a playfieldem	118
7.8	Podsumowanie wiadomości o rejestrach obsługi spriteów	118
7.9	Przykład procedury obsługującej spritea	119
Rozdział VIII		
Obsługa dźwięku		122
8.1	Wstęp	122
8.2	Przetworniki C/A	124
8.3	Odtwarzanie dźwięku	124
8.3.1	Określanie adresu sampla w pamięci — AUDxLCH i AUDxLCL	124
8.3.2	Określanie długości danych do odtworzenia — AUDxLEN	124
8.3.3	Ustalanie głośności kanału — AUDxVOL	125
8.3.4	Określanie tempa odtwarzania dźwięku — AUDxPER	126
8.3.5	Odtwarzanie dźwięku	127
8.4	Modulacja dźwięku	127

8.5	Filtr dolnoprzepustowy	128
8.6	Tryb „ręczny”	128
8.7	Wykorzystanie dźwięku w praktyce	129
8.8	Odtwarzanie modułów muzycznych	131

Rozdział IX

Kontrola systemu i obsługa interfejsów	133
9.1 Wprowadzenie	133
9.2 Priorytety wizji	133
9.2.1 Priorytety spriteów	133
9.2.2 Grupowanie spriteów	134
9.2.3 Ustawianie priorytetów pomiędzy spriteami a playfieldem	134
9.3 Wykrywanie kolizji	135
9.4 Pozycja promienia, jej sprawdzanie i związek z animacją	136
9.5 Przerwania i stany wyjątkowe	138
9.5.1 Stany wyjątkowe i wektory	138
9.5.2 Przykłady i objaśnienia stanów wyjątkowych	140
9.5.2.1 Instrukcje nielegalne i niezaimplementowane	141
9.5.2.2 Naruszenie uprzywilejowania	142
9.5.2.3 Śledzenie	142
9.5.2.4 Instrukcje TRAP #x	142
9.5.2.5 Instrukcja TRAPV	143
9.5.2.6 Instrukcja CHK	144
9.5.2.7 Dzielenie przez zero	145
9.5.3 Przerwania	145
9.5.3.1 Omówienie i przykłady inicjacji przerwań	148
9.6 Układy CIA	149
9.6.1 CIA-A	150
9.6.2 Przerwanie układu CIA-A	154
9.6.3 Klawiatura	155
9.6.4 CIA-B	159
9.6.5 Kontrola stacji dysków	163
9.6.5.1 Pojemność dyskietki, a kod MFM	163
9.6.5.2 Obsługa kontrolera stacji dysków	165
9.6.5.3 Użytkowanie DMA dysku	172
9.6.5.4 Przykładowa procedura odczytu z dysku	174
9.6.6 Bootblock	177
9.7 Obsługa urządzeń sterowniczych	180
9.7.1 Myszka i trackball	180
9.7.2 Joystick	183
9.7.3 Pióro świetlne	184
9.8 Podstawy kontroli systemu operacyjnego	184
9.8.1 Biblioteki	185
9.8.2 Otwieranie i zamykanie bibliotek	186
9.8.3 Rezerwacja pamięci	187
9.8.4 Zwalnianie zarezerwowanej pamięci	188
9.8.5 Włączanie i wyłączanie multitaskingu	188
9.8.6 Operacje na plikach	189
9.8.6.1 Otwarcie i zamknięcie pliku	189
9.8.6.2 Odczyt i zapis danych	190

9.9	Współpraca z systemem operacyjnym	191
9.9.1	Układy CIA i przerwania	191
9.9.2	Rejestr D0	192
9.9.3	Copperlista systemowa	192
9.9.4	Pamięć	192
Rozdział X		
	Grafika przestrzenna	194
10.1	Wstęp	194
10.2	Trzy wymiary	194
10.2.1	Przekształcenia współrzędnych w przestrzeni	195
10.2.1.1	Obroty	195
10.2.1.2	Przesunięcia	196
10.2.2	Rzutowanie perspektywiczne	196
10.3	Realizacja przekształceń trójwymiarowych w assemblerze	197
10.3.1	Przesunięcia	199
10.3.2	Perspektywa	199
10.3.3	Obroty	200
10.4	Przyspieszenie podprogramów obliczających	202
10.5	Wizualizacja obiektów	204
10.5.1	Reprezentacja punktowa	204
10.5.2	Reprezentacja szkieletowa	205
10.5.3	Reprezentacja szkieletowa z usuniętymi niewidocznymi liniami	206
10.5.4	Reprezentacja grafiki wypełnianej	208
10.5.4.1	Wektorówka „wypukła”	209
10.5.4.2	Wektorówka „niewypukła”	209
10.6	Kompletna procedura prezentacji wypełnianej grafiki przestrzennej	211
10.7	Clipping	220
Rozdział XI		
	Optymalizacja szybkości działania programów	223
11.1	Zastępowanie instrukcji	223
11.2	Sztuczki i triki	226
11.3	Optymalizacja czasowa pętli	226
11.4	Szybkie czyszczenie i wypełnianie pamięci	227
11.5	Główne zasady pisania szybkich procedur	227
Rozdział XII		
	Układy AGA	229
12.1	Co nowego oferują układy AGA?	229
12.2	Nowe tryby graficzne	230
12.3	Omówienie nowych możliwości	231
12.3.1	Bitplany	231
12.3.2	Spritey	233
12.3.2.1	Poruszanie spritem z precyzją do 1/4 punktu	235
12.3.2.2	Zmiana palety kolorów spriteów	235
12.4	Tabela adresowania rejestrów kolorów	237
12.5	Kompatybilność	238
12.6	Kolizje	238

12.7	Monitory	238
12.8	Przełączanie z 15 kHz na 31 kHz	238
12.9	Jak wykryć układy AGA ?	239
Dodatek I		
	Lista rejestrów w porządku alfabetycznym	240
Dodatek II		
	Lista rejestrów w kolejności adresowej	261
Dodatek III		
	Omówienie pozostałych rozkazów Motoroli 68000	269
Dodatek IV		
	Tablice czasu wykonywania poszczególnych instrukcji MC 68000	283
Dodatek V		
	Opis programu ASM ONE	292
Dodatek VI		
	Skorowidz rozkazów MC 68000	304

Rozdział I

Wprowadzenie

1.1 Systemy liczenia

W codziennym życiu mamy do czynienia z dziesiętnym (decymalnym) systemem zapisu liczb. Oznacza to, że do zapisu dowolnej liczby używamy dziesięciu cyfr: 0, 1, 2, 3, 4, 5, 6, 7, 8 i 9. To, że przyzwyczailiśmy się do niego powoduje, że nie zastanawiamy się co oznacza 143. Jednak teraz podejmiemy do tego zagadnienia z innej strony. Umożliwi to nam łatwiejsze zrozumienie sposobu zapisu liczb, jaki funkcjonuje w komputerach.

$$143 = 100 + 40 + 3 = 1 \cdot 100 + 4 \cdot 10 + 3 \cdot 1 = 1 \cdot 10^2 + 4 \cdot 10^1 + 3 \cdot 10^0$$

Jak można zauważyć, liczba dziesiętna to suma iloczynów cyfr, przez kolejne potęgi liczby 10. Niestety, mikroprocesor rozpoznaje liczby zapisane za pomocą zer i jedynek. System taki został przyjęty dlatego, że elektroniczne układy komputera mogą znajdować się w dwóch stanach:

- 0 — wyłączony (prąd nie płynie),
- 1 — włączony (prąd płynie).

Jak zapisać liczbę w systemie binarnym (czyli dwójkowym)? Postępujemy analogicznie jak w systemem dziesiętnym. *Liczba binarna jest sumą iloczynów cyfr (0 lub 1) przez kolejne potęgi liczby 2.*

$$\begin{aligned} 143 &= 1 \cdot 128 + 0 \cdot 64 + 1 \cdot 32 + 0 \cdot 16 + 1 \cdot 8 + 0 \cdot 4 + 1 \cdot 2 + 1 \cdot 1 = \\ &= 1 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 10101011 \end{aligned}$$

Używanie systemu binarnego byłoby dla nas katorgą, więc wykorzystamy inny system, zwany szesnastkowym (heksadecymalnym). Pewnie zapytasz po co? Otóż ma on zalety których nie posiada żaden inny. Ale o tym później. Liczba heksadecymalna, jak można się domyślić, jest sumą iloczynów cyfr przez potęgi liczby 16. W systemie tym mamy cyfry: 0, 1, 2, 3, ..., 8, 9, a, b, c, d, e, f. Jak można zauważyć, a(hex)=10(dec), a f(hex)=15(dec). Wszystkie informacje znajdujące się w pamięci komputera są liczbami. Grafika, muzyka, teksty, programy to wszystko są liczby w postaci binarnej, które oczywiście można przedstawić za pomocą innych systemów liczbowych. Na przykład słowo "TEKST" składa się z następujących liczb (zapisanych w tzw. kodzie ASCII):

T	E	K	S	T
#084	#069	#075	#083	#084

...	11	12	13	14	15	16	...
...	76543210	76543210	76543210	76543210	76543210	76543210	...

Liczby u góry są adresami poszczególnych komórek (bajtów) pamięci, a cyfry u dołu są numerami poszczególnych bitów. Pamiętajmy, że są one numerowane rosnąco w lewo.

1.4 Co to jest assembler i kod maszynowy

Czym dokładnie jest kod maszynowy? Jest to po prostu język programowania. Język „dziwny” dla ludzi programujących tylko w BASICu czy Pascalu. A dlaczego dziwny? Jednej instrukcji PRINT BASICA, odpowiada kilkadziesiąt instrukcji kodu maszynowego. Świadczy to o tym, że przy prostych programach nie opłaca się go używać. Osobiście do sprawdzania nowych pomysłów używam BASICA. Jednak przy programach, które mają coś szybko obliczyć lub narysować, kod maszynowy jest bezkonkurencyjny. Daje on nam władzę nad całym komputerem, umożliwia wykorzystanie jego możliwości do maksimum, a poza tym jest *najszybszy*. Dzieje się tak dlatego, ponieważ jest to język zrozumiały dla mikroprocesora. BASIC jest programem napisanym w kodzie maszynowym, który analizuje wpisane przez nas komendy i zleca ich wykonanie mikroprocesorowi. Proces ten można porównać do znajomości języka obcego — to oczywiste, że znacznie szybciej porozumiemy się we własnym języku, niż obcym.

Języki programowania można podzielić na trzy grupy:

- kod maszynowy (język wewnętrzny),
- assembler (język symboliczny),
- języki wyższego rzędu.

Program w kodzie maszynowym jest po prostu ciągiem zer i jedynek, które są pobierane przez mikroprocesor, a następnie wykonywane jako poszczególne rozkazy. Jednak pisanie programów jako ciągów liczbowych byłoby bardzo kłopotliwe. Stworzono więc symboliczny język programowania — assembler. Każdemu ciągowi liczb przyporządkowana jest odpowiednia instrukcja. Pojedyncza instrukcja zwana jest *mnemonikiem*, a proces pisania programu — *kodowaniem*.

Program w assemblerze jest zwykłym tekstem zapisanym w postaci kodów ASCII. Jest to tzw. *kod źródłowy* (ang. *source code*). Dopiero w procesie *asemblacji*, wykonywanym przez specjalny program (także zwany *assemblerem*), mnemoniki zostają przetworzone na kod maszynowy (inaczej *kod wynikowy*). Posługiwanie się assemblerem jest znacznie wygodniejsze od pisania w kodzie maszynowym.

Oto przykład:

Mnemonic		Kod maszynowy
MOVE.L #\$50000,D0	asemblacja →	\$203c, \$0005, \$0000
MOVE.L D0,\$20006		\$33c0, \$0002, \$0006
SWAP D0		\$4840
MOVE.L D0,\$20000		\$33c0, \$0002, \$0000

Rozdział II

Mikroprocesor MC 68000

2.1 Mikroprocesor Motorola 68000

Na początku należy się małe wyjaśnienie. Otóż będziemy się uczyć asemblera mikroprocesora *Motorola 68000*, jako podstawowego i najbardziej rozpowszechnionego. Będziemy także bazowali na przykładach dla *Amigi 500*. Lecz zainteresowanych dodatkowymi możliwościami innych modeli *Amig* (z nowymi układami graficznymi), odsyłam do innych rozdziałów, w których omówione są różnice pomiędzy „starymi” i „nowymi” układami, oraz ich nowe możliwości.

Motorola 68000 jest mikroprocesorem 16 bitowym z 24-bitową szyną adresową. Oznacza to, że pamięć (przestrzeń adresowa) przez nią obsługiwana, może mieć 2^{24} adresów (16 777 216). Dane które może interpretować i wykonywać na nich operacje matematyczne (oprócz dzielenia i mnożenia) są 32-bitowe. Inaczej mówiąc, posiada możliwość wykonywania operacji na liczbach w zakresie od 0 do 4 294 967 295. A co z liczbami ujemnymi? Poradzono sobie z tym w taki sposób: najstarszy bit danej określa, czy liczba jest dodatnia, czy ujemna. Na przykład:

#-1	=	\$ff	bajt
#-1	=	\$ffff	słowo
#-1	=	\$ffffffff	długie słowo
#-2	=	\$fe	bajt
#-2	=	%11111110	bajt

Jak widać, jest to sposób bardzo podobny do normalnego (dodatniego) zapisu liczb.

#1	=	%00000001
#-2	=	%11111110

Na tej podstawie można wywnioskować, że liczba ujemna jest wartością bezwzględną pomniejszoną o 1 i zanegowaną (tzn. 0 zamieniane jest na 1, a 1 na 0).

Dla przykładu zapiszemy teraz binarnie liczbę -7.

$$\begin{array}{rcl}
 \#7 & = & \%00000111 \\
 & & \underline{-\%00000001} \\
 \text{neg} & & \%00000110 \\
 & & \underline{ \%11111001} \\
 & & \%11111001 = \#-7
 \end{array}$$

Bardzo często mikroprocesory oceniane są pod względem liczby **rejestrów**. Są to małe bloki pamięci (zawarte w mikroprocesorze), w których mikroprocesor przechowuje potrzebne dane. Wydawałoby się, że nie jest to potrzebne, ale to tak jak z kieszeniami w ubraniu. Zamiast iść po jakiś drobiazg, sięgamy po prostu do kieszeni. Dzięki rejestrom, operacje na liczbach są wykonywane o wiele szybciej niż operacje wykonywane bezpośrednio na komórkach pamięci.

MC 68000 posiada 8 rejestrów danych (oznaczonych kolejno: D0, D1, D2, ..., D6, D7) używanych jako 8, 16 lub 32-bitowe rejestry służące głównie do wykonywania działań arytmetycznych, chwilowego przechowywania danych, indeksowania pamięci. Oprócz rejestrów danych, istnieje 8 rejestrów adresowych, oznaczonych A0, A1, A2, ..., A6, A7. Rejestry te są używane jako 16 i 32-bitowe i służą do czasowego przechowywania adresów lub oznaczają miejsca skąd lub dokąd mają być zapisywane dane. Rejestr A7 jest tutaj wyjątkiem. Służy on jako tzw. **wskaznik stosu**.

Czym jest stos? Jest to fragment pamięci przeznaczony do tymczasowego przechowywania danych. Stos może przypominać stos kartek. Możemy zobaczyć zawartość kartki tylko na wierzchołku stosu. Aby zobaczyć co jest pod spodem, należy zdjąć kartkę z góry. Stos może być wykorzystywany zarówno przez użytkownika, jak i przez mikroprocesor. Użytkownik może odkładać na niego chwilowe dane, a najczęściej zawartości rejestrów. Natomiast mikroprocesor odkłada na stos licznik programu (PC — czyli adres aktualnie wykonywanej instrukcji) przy skoku do podprogramu lub przy obsłudze procedury **przerwań** (procedurą nazywamy fragment programu wykonujący jakąś konkretną czynność). Przerwaniom poświęcony jest osobny rozdział tej książki.

Następnym ważnym rejestrem jest tzw. **rejestr systemowy**, składający się z dwóch bajtów. Rejestr ten jest statusem mikroprocesora.

bit	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
	T	.	.	S	.	*	*	*	.	.	.	X	N	Z	V	C
	Bajt systemowy								Bajt użytkownika							

Na razie nie będziemy omawiać znaczenia bitów w bajcie systemowym (jest on opisany w rozdziale o stanach wyjątkowych).

Bajt użytkownika zawiera zestaw bitów zwanych **kodami warunków**. Ich zadaniem jest scharakteryzowanie wyniku ostatnio wykonanej instrukcji.

Tak więc:

- bit 0 — bit przeniesienia (ang. *Carry*) — jest ustawiany lub zerowany w operacjach arytmetycznych i logicznych, jest również używany w przesunięciach i obrotach bitów,
- bit 1 — bit przepełnienia (ang. *oVerflow*) — jest ustawiany, gdy wynik operacji nie może być prawidłowo przedstawiony,
- bit 2 — bit zera (ang. *Zero*) — jest ustawiany, gdy wynikiem operacji było zero, w przeciwnym wypadku jest kasowany („gaszony”),
- bit 3 — bit ujemności (ang. *Negative*) — jest ustawiany, gdy wynikiem ostatniej operacji była liczba ujemna i kasowany, gdy liczba była dodatnia.
- bit 4 — bit rozszerzenia (ang. *eXtend*) — jest on bitem analogicznym do C (*Carry*), ale jest ustawiany przez inne instrukcje, służy do operacji na liczbach wielokrotnej precyzji.

2.2 Rozkazy Motoroli 68000

Rozkazy Motoroli 68000 możemy podzielić na następujące grupy:

- instrukcje przekazywania danych: EXG, LEA, LINK, MOVE, MOVEM, MOVEP, MOVEQ, PEA, SWAP, UNLK,
- instrukcje arytmetyczne: ADD, ADDA, ADDI, ADDQ, ADDX, CLR, CMP, CMPA, CMPI, CPM, DIVS, DIVU, EXT, MULS, MULU, NEG, NEGX, SUB, SUBA, SUBI, SUBQ, SUBX, TAS, TST,
- instrukcje logiczne: AND, ANDI, OR, ORI, EOR, EORI, NOT,
- instrukcje przesunięć i obrotów: ASL, ASR, LSL, LSR, ROL, ROR, ROXL, ROXR,
- instrukcje manipulacji bitami: BTST, BSET, BCLR, BCHG,
- instrukcje BCD: ABCD, SBCD, NBCD,
- instrukcje sterowania programem: Bxx, DBxx, Sxx, BSR, JSR, JMP, RTS, RTR,
- instrukcje kontroli systemu: MOVE USP, RESET, RTE, STOP, CHK, TRAPV, TRAP.

Ponieważ każda instrukcja Motoroli 68000 zabiera co najmniej 2 bajty pamięci, żadna instrukcja nie może rozpoczynać się od nieparzystego adresu. W wypadku wystąpienia takiej sytuacji program asemblerujący informuje nas o tym, wyświetlając stosowny komunikat. Również mikroprocesor nie może operować danymi o długości słowa lub dłużego słowa, pod nieparzystym adresem.

Istnieją instrukcje kontrolujące proces asemblacji, które umożliwiają wyrównanie do parzystego adresu. Są to EVEN lub CNOP (nie są to instrukcje mikroprocesora, lecz instrukcje kontrolujące proces asemblacji w danym programie asemblerującym).

2.3 Tryby adresowania

Teraz nadszedł czas na poznanie ogólnej zasady działania pierwszej z podstawowych instrukcji.

MOVE.X <źródło>, <przeznaczenie>

Instrukcja ta służy do przenoszenia danych ze źródła (*operandu źródłowego*), do miejsca przeznaczenia (*operandu przeznaczenia*). Dla adresów operandu źródłowego i docelowego, używa się wspólnego określenia — *adres efektywny* (skrót <ea>). Operandy możemy określić w różny sposób, za pomocą odpowiednich *trybów adresowania*, które określają w jaki sposób mają być obliczane adresy źródła i przeznaczenia.

Zamiast znaku "X" musimy podać jedną z trzech liter: "B", "W" lub "L". Oznaczać ona będzie, na ilu bitach będzie wykonywana operacja. Podając np. "B", spowodujemy, że przeniesiony zostanie jeden bajt (ang. *Byte*). "W" to słowo (ang. *Word*), a "L" — długie słowo (ang. *Long word*).

Tryb adresowania bezpośredniego rejestru danych

Format:

Dn

gdzie:

n — numer rejestru.

Przykład:

MOVE.W D0, D7

Zawartość rejestru D0 zostanie skopiowana do D7. Przeniesione zostanie 16 bitów (słowo).

Tryb adresowania bezpośredniego rejestru adresowego

Format:

An

gdzie:

n — numer rejestru.

Przykład:

MOVE.L A2, D1

Zawartość rejestru A2, zostanie przeniesiona do D1. Przeniesione zostanie długie słowo. Nie można jednak w tym wypadku przesłać danej o rozmiarze bajtu.

Tryb adresowania pośredniego rejestrem adresowym**Format:****(An)**

gdzie:

n — numer rejestru.**Przykład:**`MOVE.W (A5), D0`

Zawartość pamięci pod adresem zawartym w A5, zostanie przeniesiona do rejestru D0. Na przykład jeśli w A5 znajduje się adres \$50000, to słowo spod tego adresu byłoby przeniesione do D0.

Tryb adresowania pośredniego rejestrem adresowym z postinkrementacją (zwiększeniem zawartości rejestru)**Format:****(An)+**

gdzie:

n — numer rejestru.**Przykład:**`MOVE.L (A3)+, D1`

Zawartość pamięci pod adresem w A3, zostanie przeniesiona do rejestru D1, przy czym zawartość A3 zwiększy się o długość pobranych danych (w naszym wypadku o 4 ponieważ przesłaliśmy długie słowo — 4 bajty). W wypadku przesłania bajtu, zawartość A3 zwiększyłaby się o 1, a jeśli słowo (dwa bajty), to o 2.

Przypuśćmy, że pod adresem \$20000 mamy jakieś słowo danych (np. \$ff23), a rejestr A0 zawiera wartość \$20000. Po wykonaniu instrukcji `MOVE.W (A0)+, D1`, w rejestrze D0 znajdzie się wartość \$ff23 (przeniesione zostało słowo), a rejestr adresowy A0 zawierać będzie adres \$20002.

Tryb adresowania pośredniego rejestrem adresowym z predekrementacją (zmniejszeniem zawartości rejestru)**Format:****-(An)**

gdzie:

n — numer rejestru.**Przykład:**`MOVE.W -(A5), D4`

Zawartość pamięci pod adresem w A5, zostanie przeniesiona do rejestru D4, przy czym zawartość A5 zmniejszy się o 2 (ponieważ w przykładzie przenosiłoby 2 bajty — słowo).

Tryb adresowania pośredniego rejestrem adresowym z przesunięciem**Format:****x(An)**

gdzie:

n — numer rejestru,**x** — 16-bitowe przesunięcie.**Przykład:**`MOVE.L 6(A0), D0`

Spod adresu będącego sumą zawartości rejestru A0 i 16-bitowego przesunięcia, przesłane zostanie długie słowo do rejestru D0. Przypuśćmy, że w rejestrze A0 mamy adres \$40000. W wypadku wykonania instrukcji `MOVE.L 8(A0), D0`, w rejestrze D0 znajdzie się długie słowo spod adresu $\$40000 + 8 = \40008 .

Tryb adresowania pośredniego rejestrem adresowym z indeksem i przesunięciem**Format:****x(An,Dm.w)****x(An,Dm.l)****x(An,Am.w)****x(An,Am.l)**

gdzie:

n, m — numer rejestrów,**x** — 8-bitowe przesunięcie.**Przykład:**`MOVE.B 3(A2, D0.w), D2`

Spod adresu, który jest sumą zawartości A2, 8-bitowego przesunięcia oraz słowa w D0 zostanie przesłany bajt, który będzie załadowany do D2.

Tryb adresowania absolutnego krótkiego**Format:****adres**

gdzie:

adres — 16-bitowy adres.**Przykład:**`MOVE.W $200.w, D1`

Do rejestru D1, przesłana zostanie zawartość komórek \$200 i \$201, jako słowo. Adres może być jedynie w obrębie słowa (maksymalnie \$ffff).

Tryb adresowania absolutnego długiego

Format:**adres**

gdzie:

adres — 24-bitowy adres.**Przykład:**`MOVE.B $22320, D0`

Spod adresu \$22320, do D0 przesłany zostanie bajt. Adres może być dowolny.

Tryb adresowania licznikiem programu z przesunięciem

Format:**x(PC)**

gdzie:

x — 16-bitowe przesunięcie.**Przykład:**`MOVE.W 2(PC), D0`

Spod adresu będącego sumą PC i 16-bitowego przesunięcia, przesłane zostanie słowo do rejestru D0.

Tryb adresowania licznikiem programu z indeksem i przesunięciem

Format:**x(PC,Dn.w)****x(PC,Dn.l)****x(PC,An.w)****x(PC,An.l)**

gdzie:

n — numer rejestru,**x** — 8-bitowe przesunięcie.**Przykład:**`MOVE.B 8(PC, D2.l), D2`

Spod adresu, który jest sumą PC, 8-bitowego przesunięcia oraz długiego słowa w D2 zostanie przesłany bajt, który będzie załadowany do D2.

Tryb adresowania natychmiastowy

Format:

#x

gdzie:

x — liczba.

Przykład:

MOVE.L #\$ffff,D0

Zawartością rejestru D0 będzie liczba \$0000ffff.

Tryb adresowania rejestru statusowego (SR i CCR)

Ten tryb adresowania określa, że operacja ma być przeprowadzona na rejestrze statusowym (SR — ang. *Status Register*). W przypadku podania rozmiaru operacji jako słowa, SR odnosi się do całego rejestru statusowego. Gdy podamy jako rozmiar bajt, to będziemy odnosić się do bajtu użytkownika w tym rejestrze. Bajt użytkownika określa także CCR.

Do instrukcji umożliwiających wykorzystanie tego trybu adresowania, należą: ANDI, EORI, ORI oraz MOVE <ea>,SR.

2.4 Kod BCD

Istnieje specyficzny format zapisu liczb binarnych zwany kodem BCD (szeroko używany w kalkulatorach).

Kod BCD opiera się na kodzie heksadecymalnym z tym, że nie są używane kombinacje od 10 do 15 (czyli od \$a do \$f). Każda czwórka bitów reprezentuje jedną cyfrę dziesiętną, tak więc w jednym bajcie przechowywane są dwie cyfry dziesiętne. Umożliwia on na bardzo szybką zmianę liczby na postać znakową.

Motorola posiada trzy rozkazy umożliwiające operacje na liczbach zapisanych w tym kodzie.

2.5 Błędne rozkazy a program asemblerowy

Napisano już wiele assemblerów działających na Amidze. Praktycznie skończyły się już czasy, że pomyłka w składni rozkazu lub niewłaściwy tryb adresowania, powodowały duże kłopoty (np. „GURU MEDITAION”). Większość dostępnych assemblerów (np. TRASH'M ONE 1.6, ASM-ONE) są rozbudowanymi „kombajnami”, które w procesie asemblacji starają się te błędy skorygować. Gdy na przykład napiszemy:

ADD.L #10,A0

to w procesie asemblacji zostanie to zamienione na właściwy rozkaz. Po prostu asembler traktuje tę instrukcję jako niewykonywalną dla procesora i zamienia ją na właściwą:

```
ADDA.L #10,A0.
```

Uwalnia to programistę od dokładnego zapamiętania znaczenia wszystkich rozkazów. Wystarczy podawać mniej więcej jak rozkaz powinien „brzmieć”.

2.6 Pisanie procedur wykonujących operacje arytmetyczne

Teraz nauczymy się wykonywać podstawowe operacje arytmetyczne. Na początek dokładnie poznamy rozkazy przesyłania, dodawania i odejmowania, oraz mnożenia i dzielenia.

Rozkaz MOVE (Move data from source to destination — przesłanie danej)

Składnia:

```
MOVE <ea>,<ea>
```

Atrybuty:

rozmiar: B, W, L

Działanie:

Instrukcja MOVE przesyła zawartość operandu źródłowego, do operandu przeznaczenia, przy czym zawartość operandu źródłowego pozostaje niezmienną. Rozmiar operacji może być określony jako bajt, słowo lub długie słowo.

Kody warunków:

X — nie zmieniany,

N — ustawiany, gdy najstarszy bit przesyłanej danej jest równy jeden, w przeciwnym wypadku zerowany,

Z — ustawiany jeśli dana wynosiła zero, w przeciwnym wypadku zerowany,

V — zawsze zerowany,

C — zawsze zerowany.

Dozwolone tryby adresowania (dla operandu źródłowego):

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
tak	tak	tak	tak	tak	tak	tak
X.w	X.l	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	tak	tak	tak	nie	nie

Atrybuty:

rozmiar: W, L

Działanie:

Instrukcja MOVEM umożliwia szybkie przesłanie grupy rejestrów do lub z pamięci. Rozmiar operacji może być określony jako słowo lub długie słowo. W przypadku przesyłania słowa danych z pamięci do rejestrów, zostają one automatycznie rozszerzone do długiego słowa.

Instrukcja ta jest używana przede wszystkim do tymczasowego przechowywania zawartości rejestrów na stosie.

Listę rejestrów podajemy w następujący sposób:

- oddzielając każdy rejestr znakiem "/":
MOVEM.L D0/D1/D2/A0/A1/A2/A6, -(A7)
- oddzielając kolejne rejestry znakiem "-" (od - do):
MOVEM.L D0-D2/A0-A2/A6 = MOVEM.L D0/D1/D2/A0/A1/A2/A6, -(A7)

Kody warunków:

X — nie zmieniany,
N — nie zmieniany,
Z — nie zmieniany,
V — nie zmieniany,
C — nie zmieniany.

Dozwolone tryby adresowania (przesyłanie rejestrów do pamięci):

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
nie	nie	tak	nie	tak	tak	tak
X.w	X.l	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	nie	nie	nie	nie	nie

Dozwolone tryby adresowania (przesyłanie pamięci do rejestrów):

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
nie	nie	tak	tak	nie	tak	tak
X.w	X.l	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	tak	tak	nie	nie	nie

Rozkaz ADD (Add binary — dodawanie binarne)**Składnia:**

ADD <ea>,Dn

ADD Dn,<ea>

Atrybuty:

rozmiar: B, W, L

Działanie:

Instrukcja ADD dodaje operand źródłowy, do operandu przeznaczenia. Wynik operacji zostaje zapisany w miejscu przeznaczenia. Rozmiar operacji może być określony jako bajt, słowo lub długie słowo.

Kody warunków:

- X — ustawiany gdy wystąpi przeniesienie, w przeciwnym wypadku kasowany,
- N — ustawiany wtedy, gdy najstarszy bit przesyłanej danej równy jest jeden, w przeciwnym wypadku jest zerowany,
- Z — ustawiany gdy wynik operacji jest zerowy, w przeciwnym wypadku zerowany,
- V — ustawiany, gdy wystąpi nadmiar, w przeciwnym wypadku zerowany,
- C — ustawiany gdy wystąpi przeniesienie, w przeciwnym wypadku kasowany.

Uwaga:

Przeniesieniem nazywamy sytuację, w wyniku której zwiększa się liczba cyfr przypadających na wynik operacji (np. dodawania). Oto prosty przykład:

brak przeniesienia	przeniesienie
$\begin{array}{r} 8 \\ + 1 \\ \hline 9 \end{array}$	$\begin{array}{r} 9 \\ + 4 \\ \hline 13 \end{array}$

Taka sytuacja zachodzi w przypadku dodawania do siebie np. dwóch liczb o długości bajtu, których wynikiem jest liczba 9-cio bitowa. Dla uproszczenia, w przykładzie zaniedbujemy liczby ujemne.

$$\begin{array}{r} \% 10010011 \\ + \% 10101101 \\ \hline \% 101000000 \end{array}$$

W przypadku, gdy do prawidłowego zapisania wyniku operacji nie jest wystarczający zadeklarowany rozmiar operandów i potrzebny jest dodatkowy bit, następuje przeniesienie.

Wykrywanie i obsługa takich sytuacji umożliwia wykonywanie operacji dodawania za pomocą odpowiednich rozkazów (np. ADDX) na liczbach o dowolnej długości (np. 2000 bajtów).

Dozwolone tryby adresowania (gdy adres efektywny określa operand źródłowy):

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
tak	tak	tak	tak	tak	tak	tak
X.w	X.l	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	tak	tak	tak	nie	nie

Dozwolone tryby adresowania (gdy adres efektywny określa operand przeznaczenia):

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
nie	nie	tak	tak	tak	tak	tak
X.w	X.l	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	nie	nie	nie	nie	nie

Przykład:

ADD.W 2(A0,D2.1),D1

Słowo spod adresu wskazywanego przez sumę zawartości rejestrów A0 i D2 oraz 2 zostanie dodane do młodszego słowa rejestru D1. Wynik zostanie zapisany w młodszym słowie rejestru D1.

Rozkaz ADDI (Add immediate — dodawanie natychmiastowe)

Składnia:

ADDI #<dana>,<ea>

Atrybuty:

rozmiar: B, W, L

Działanie:

Instrukcja ADDI dodaje daną natychmiastową, czyli liczbę podaną w instrukcji, do operandu przeznaczenia. Wynik operacji zostaje zapisany w miejscu przeznaczenia. Rozmiar operacji może być określony jako bajt, słowo lub długie słowo.

Kody warunków:

- X — ustawiany, gdy wystąpi przeniesienie, w przeciwnym wypadku kasowany,
- N — ustawiany, gdy najstarszy bit przesyłanej danej równy jest jeden, w przeciwnym wypadku jest zerowany,
- Z — ustawiany, gdy wynik operacji jest zerowy, w przeciwnym wypadku zerowany,
- V — ustawiany, gdy wystąpi nadmiar, w przeciwnym wypadku zerowany,

C — ustawiany gdy wystąpi przeniesienie, w przeciwnym wypadku kasowany.

Dozwolone tryby adresowania:

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
tak	nie	tak	tak	tak	tak	tak
X.w	X.l	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	nie	nie	nie	nie	nie

Przykład:

ADDI.L #\$2800, (A0)

Zawartość długiego słowa pod adresem wskazywanym przez A0 zostanie zwiększona o \$2800.

Rozkaz ADDQ (Add quick — dodawanie szybkie)

Składnia:

ADDQ #<dana>, <ea>

Atrybuty:

rozmiar: B, W, L

Działanie:

Instrukcja ADDQ dodaje daną natychmiastową do operandu przeznaczenia, przy czym dana musi wynosić od 1 do 8. Wynik zostaje zapisany w miejscu przeznaczenia.

Rozmiar operacji może być określony jako bajt, słowo lub długie słowo.

Kody warunków:

X — ustawiany gdy wystąpi przeniesienie, w przeciwnym wypadku kasowany,

N — ustawiany wtedy, gdy najstarszy bit przesyłanej danej równy jest jeden, w przeciwnym wypadku jest zerowany,

Z — ustawiany gdy wynik operacji jest zerowy, w przeciwnym wypadku zerowany,

V — ustawiany, gdy wystąpi nadmiar, w przeciwnym wypadku zerowany,

C — ustawiany gdy wystąpi przeniesienie, w przeciwnym wypadku kasowany.

Dozwolone tryby adresowania:

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
tak	tak	tak	tak	tak	tak	tak
X.w	X.l	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	nie	nie	nie	nie	nie

Przykład:

```
ADDQ.W    #4, (A0,D0.W)
```

Zawartość słowa pod adresem wskazywanym przez sumę A0 i młodszego słowa D0 zostanie zwiększona o 4.

Rozkaz SUB (Subtract binary — odejmowanie binarne)**Składnia:**

```
SUB <ea>,Dn
```

```
SUB Dn,<ea>
```

Atrybuty:

rozmiar: B, W, L

Działanie:

Instrukcja SUB odejmuje operand źródłowy od operandu przeznaczenia. Wynik zostaje zapisany w miejscu przeznaczenia. Rozmiar instrukcji może być określony jako bajt, słowo lub długie słowo. Istnieją dwie formy tej instrukcji:

- odjęcie operandu określonego adresem efektywnym od rejestru danych:

```
SUB.L $10232,D0  
SUB.L (A0),D0
```
- odjęcie zawartości rejestru danych od operandu określonego adresem efektywnym:

```
SUB.L D0,$10232  
SUB.L D0,(A0)
```

W wypadku użycia rejestru adresowego jako operandu źródłowego możliwe jest tylko dla operacji o rozmiarze słowa lub długiego słowa.

Kody warunków:

- X — ustawiany gdy wystąpi pożyczka, w przeciwnym wypadku kasowany,
- N — ustawiany gdy wynik jest ujemny, w przeciwnym wypadku jest zerowany,
- Z — ustawiany gdy wynik operacji jest zerowy, w przeciwnym wypadku zerowany,
- V — ustawiany, gdy wystąpi nadmiar, w przeciwnym wypadku zerowany,
- C — ustawiany gdy wystąpi pożyczka, w przeciwnym wypadku kasowany.

Uwaga:

Pożyczka jest odwrotnością przeniesienia, tyle że występuje przy odejmowaniu.

Jak wiadomo, przy odejmowaniu pisemnym, gdy odejmujemy większą liczbę od mniejszej, „pożycza się” 1 od liczby stojącej bezpośrednio na lewo. Właśnie tę sytuację „pożyczania” z najstarszego bitu danych w celu wykonania operacji odejmowania — nazywamy pożyczką.

Dozwolone tryby adresowania (gdy adres efektywny określa operand źródłowy):

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
tak	tak	tak	tak	tak	tak	tak
X.w	X.1	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	tak	tak	tak	nie	nie

Dozwolone tryby adresowania (gdy adres efektywny określa operand przeznaczenia):

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
nie	nie	tak	tak	tak	tak	tak
X.w	X.1	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	nie	tak	tak	nie	nie

Przykład:

SUB.W (A1,D0.w),D1

Słowo spod adresu wskazywanego przez sumę zawartości rejestrów A1 i młodszego słowa D0 oraz 2 zostanie odjęte od młodszego słowa rejestru D1. Wynik zostanie zapisany w młodszym słowie rejestru D1.

Rozkaz SUBI (Subtract immediate — odejmowanie natychmiastowe)

Składnia:

SUBI #<dana>,<ea>

Atrybuty:

rozmiar: B, W, L

Działanie:

Instrukcja SUBI odejmuje daną natychmiastową, czyli liczbę zawartą w instrukcji, od operandu przeznaczenia. Wynik zostaje zapisany w miejscu przeznaczenia. Rozmiar operacji jest dowolny.

Kody warunków:

- X — ustawiany gdy wystąpi pożyczka z najstarszego bitu, w przeciwnym wypadku kasowany,
- N — ustawiany gdy wynik jest ujemny, w przeciwnym wypadku jest zerowany,
- Z — ustawiany gdy wynik operacji jest zerowy, w przeciwnym wypadku zerowany,
- V — ustawiany, gdy wystąpi nadmiar, w przeciwnym wypadku zerowany,

C — ustawiany gdy wystąpi pożyczka najstarszego bitu, w przeciwnym wypadku kasowany.

Dozwolone tryby adresowania:

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
tak	nie	tak	tak	tak	tak	tak
X.w	X.1	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	nie	nie	nie	nie	nie

Przykład:

SUBI.W #1,D0

Zawartość młodszego słowa rejestru D0 zostanie zmniejszona o 1.

Rozkaz SUBQ (Subtract quick — odejmowanie szybkie)

Składnia:

SUBQ #<dana>,<ea>

Atrybuty:

rozmiar: B, W, L

Działanie:

Instrukcja SUBQ odejmuje daną natychmiastową od operandu przeznaczenia, przy czym dana musi wynosić od 1 do 8. Wynik zostaje zapisany w miejscu przeznaczenia. Rozmiar operacji może być określony jako bajt, słowo lub długie słowo.

Kody warunków:

- X — ustawiany gdy wystąpi pożyczka z najstarszego bitu, w przeciwnym wypadku kasowany,
- N — ustawiany gdy wynik jest ujemny, w przeciwnym wypadku jest zerowany,
- Z — ustawiany gdy wynik operacji jest zerowy, w przeciwnym wypadku zerowany,
- V — ustawiany, gdy wystąpi nadmiar, w przeciwnym wypadku zerowany,
- C — ustawiany gdy wystąpi pożyczka z najstarszego bitu, w przeciwnym wypadku kasowany.

Dozwolone tryby adresowania:

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
tak	tak	tak	tak	tak	tak	tak
X.w	X.1	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	nie	nie	nie	nie	nie

Przykład:

SUBQ.W #2, (A0)

Wartość słowa pod adresem wskazywanym przez zawartość rejestru A0 zostanie zmniejszona o 2.

Rozkaz MULU (Unsigned multiply — mnożenie bez znaku)**Składnia:**

MULU <ea>,Dn

Atrybuty:

rozmiar: W

Działanie:

Instrukcja MULU mnoży 16-bitową daną, zawartą w młodszym słowie rejestru danych przez 16-bitowy operand określony adresem efektywnym, dając 32-bitowy wynik zapisywany w rejestrze danych. Instrukcja ta działa w arytmetyce bezznakowej.

Kody warunków:

X — nie zmieniany,

N — ustawiany gdy najbardziej znaczący bit wyniku jest równy 1, w przeciwnym wypadku zerowany,

Z — ustawiany gdy wynik jest zerowy, w przeciwnym wypadku zerowany.

V — zawsze zerowany,

C — zawsze zerowany.

Dozwolone tryby adresowania:

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
tak	nie	tak	tak	tak	tak	tak
X.w	X.l	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	tak	tak	tak	nie	nie

Przykład:

MULU 2(A0,D0.l),D1

Zawartość rejestru D1 zostanie pomnożona przez słowo wskazywane przez sumę zawartości rejestrów A0, D0 i 2.

Rozkaz MULS (Signed multiply — mnożenie ze znakiem)**Składnia:**

MULS (ea>,Dn

Atrybuty:

rozmiar: W

Działanie:

Instrukcja MULS mnoży 16-bitową daną, zawartą w młodszym słowie rejestru danych przez 16-bitowy operand określony adresem efektywnym, dając 32-bitowy wynik zapisywany w rejestrze danych. Instrukcja ta działa w arytmetyce znakowej.

Kody warunków:

X — nie zmieniany,

N — ustawiany gdy wynik jest ujemny, w przeciwnym wypadku zerowany,

Z — ustawiany gdy wynik jest zerowy, w przeciwnym wypadku zerowany,

V — zawsze zerowany,

C — zawsze zerowany.

Dozwolone tryby adresowania:

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
tak	nie	tak	tak	tak	tak	tak
X.w	X.1	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	tak	tak	tak	nie	nie

Przykład:

MULS #- \$200, D0

Zawartość rejestru D0 zostanie pomnożona przez daną natychmiastową, czyli -\$200.

Rozkaz DIVU (Unsigned Divide — dzielenie bez znaku)**Składnia:**

DIVU <ea>, Dn

Atrybuty:

rozmiar: W

Działanie:

Instrukcja DIVU dzieli 32-bitową daną, zawartą w młodszym słowie rejestru danych przez 16-bitowy operand określony adresem efektywnym. Młodsze słowo rejestru danych, to 16-bitowy wynik zapisywany, a starsze — reszta z dzielenia. Instrukcja ta działa w arytmetyce bezznakowej.

Podczas wykonywania instrukcji DIVU mogą być popełnione dwa błędy:

- próba dzielenia przez 0 — w tej sytuacji wygenerowany zostanie stan wyjątkowy,

- duża liczba jest dzielona przez liczbę małą i część całkowita ilorazu nie mieści się w 16 bitach — spełniony jest warunek nadmiaru, zostaje ustawiony bit V w rejestrze statusowym, a zawartość rejestru danych pozostaje niezmienną.

Kody warunków:

- X — nie zmieniany,
 N — ustawiany najbardziej znaczący bit wyniku jest równy 1, w przeciwnym wypadku zerowany,
 Z — ustawiany gdy wynik jest zerowy, w przeciwnym wypadku zerowany,
 V — ustawiany gdy wystąpił nadmiar, w przeciwnym wypadku zerowany,
 C — zawsze zerowany.

Dozwolone tryby adresowania:

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
tak	nie	tak	tak	tak	tak	tak
X.w	X.1	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	tak	tak	tak	nie	nie

Przykład:

DIVU 2(A0),D0

Zawartość rejestru D0 zostanie podzielona przez słowo pod adresem wskazywanym przez sumę zawartości rejestru A0 i 2.

Rozkaz DIVS (Signed Divide — dzielenie ze znakiem)

Składnia:

DIVS <ea>,Dn

Atrybuty:

rozmiar: W

Działanie:

Instrukcja DIVS dzieli 32-bitową daną, zawartą w młodszym słowie rejestru danych przez 16-bitowy operand określony adresem efektywnym. Młodsze słowo rejestru danych, to 16-bitowy wynik zapisywany, a starsze — reszta z dzielenia. Iloraz i reszta mają ten sam znak. Instrukcja ta działa w arytmetyce znakowej.

Podczas wykonywania instrukcji DIVS mogą być popełnione dwa błędy:

- próba dzielenia przez 0 — w tej sytuacji wygenerowany zostanie stan wyjątkowy,
- duża liczba jest dzielona przez liczbę małą i część całkowita ilorazu nie mieści się w 16 bitach — spełniony jest warunek nadmiaru, zostaje ustawiony bit V w rejestrze statusowym, a zawartość rejestru danych pozostaje niezmienną.

Kody warunków:

- X — nie zmieniany,
 N — ustawiany najbardziej znaczący bit wyniku jest równy 1, w przeciwnym wypadku zerowany,
 Z — ustawiany gdy wynik jest zerowy, w przeciwnym wypadku zerowany,
 V — ustawiany gdy wystąpił nadmiar, w przeciwnym wypadku zerowany,
 C — zawsze zerowany.

Dozwolone tryby adresowania:

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
tak	nie	tak	tak	tak	tak	tak
X.w	X.1	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	tak	tak	tak	nie	nie

Przykład:

DIVS #-40,D0

Zawartość rejestru D0 zostaje podzielona przez -40.

Teraz, gdy zapoznaliśmy się już z podstawowymi instrukcjami, możemy za ich pomocą wykonywać operacje matematyczne.

```

; -----
; Wzór:          X1*40+X1/8
; Wejście:       D0-X1
; Rozmiar danej: słowo

MOVE.W    D0,D1    ;przesyłamy X1 do D1
MULS      #40,D1    ;X1*40
DIVS      #8,D0     ;X1/8
ADD.W     D1,D0     ;X1*40+X1/8
RTS

; Wyjście:       D0=X1*40+X1/8
; Rozmiar danej: słowo
; -----
; Wzór:          ((X1-X2)/(Y1+Y2))*T
; Wejście:       D0-X1, D1-X2, D2-Y1, D3-Y2, D4-T
; Rozmiar danych: słowo

SUB.W     D1,D0
ADD.W     D3,D2
DIVS      D2,D0
MULS      D4,D0
RTS

; Wyjście:       D0=((X1-X2)/(Y1+Y2))*T
; Rozmiar danej: słowo
; -----

```

Myślę, że ogólna zasada tworzenia procedur obliczających nie sprawia specjalnie kłopotu. Opanowanie tej czynności daje możliwość sprawnego pisania większych programów.

Rozdział III

Hardware Amigi

3.1 Rodzaje pamięci w Amigach

Pamięć RAM w Amidze można podzielić na trzy podstawowe rodzaje:

- CHIP — pamięć dostępna zarówno dla mikroprocesora, jak i procesorów specjalizowanych. W celu przesyłania danych wykorzystuje te same kanały komunikacji (DMA), zatem gdy jest dużo danych do przesłania, muszą one czekać na swoją kolejkę, co spowalnia pracę komputera.

Uwaga: trzeba pamiętać, by wszystkie dane dla procesorów specjalizowanych znajdowały się właśnie w tym typie pamięci — w przeciwnym wypadku programy mogą działać błędnie.

- FAST — pamięć dostępna tylko dla mikroprocesora, posiada bezpośrednie połączenie z procesorem (oddzielne kanały DMA), co umożliwia rozwinięcie przez procesor największej szybkości pracy.
- SLOW — pamięć dostępna tylko dla mikroprocesora, korzystająca z tych samych kanałów DMA co pamięć CHIP.

3.2 Obszary pamięci i ich znaczenie

Oto schematyczny podział pamięci Amigi 500, 2000, 600 i 500+.

Adres	Przeznaczenie
\$000000 – \$07ffff	512 kB CHIP RAM
\$080000 – \$0fffff	512 kB CHIP RAM (Amiga 600, 500+, CDTV, 1200, 2000, CD32 Amiga 500, 2000 — opcjonalnie)
\$100000 – \$1fffff	1 MB CHIP RAM (Amiga 4000, 1200, CD32 lub 600, 500+)
\$200000 – \$9fffff	8 MB FAST RAM (opcjonalnie)
\$a00000 – \$beffff	zarezerwowane, nie używaj
\$bfd000 – \$bfdfff	CIA 8520-B

Adres	Przeznaczenie
\$bfe001 – \$bfef01	CIA 8520-A
\$c00000 – \$dbffff	SLOW RAM
\$dc0000 – \$dcffff	zegar czasu rzeczywistego
\$dff000 – \$dfffff	rejstry procesorów specjalizowanych
\$e00000 – \$e7ffff	zarezerwowane, nie używaj
\$e80000 – \$e8ffff	obszar Auto-Config
\$e90000 – \$efffff	obszar Auto-Config
\$f00000 – \$fbffff	zarezerwowane, nie używaj
\$fc0000 – \$ffffff	256 kB ROM

3.3 Procesory specjalizowane Amigi

W skład Amigi wchodzi następujące układy specjalizowane:

- Agnus (w Amigach 1200/4000/CD32, jej odpowiednik — Alice),
- Denise (w Amigach 1200/4000/CD32, jej odpowiednik — Lisa),
- Paula.

Dlaczego mówimy, że są specjalizowane? Nazywamy je tak dlatego, że każdy z nich wykonuje charakterystyczne dla siebie funkcje (np. odtwarzanie dźwięku, itd.).

AGNUS (FAT AGNUS) jest najbardziej znanym z koprocessorów. Zawiera on *Blitter* (BLOck Image TRANSFER) — układ umożliwiający operacje na prostokątnych obszarach pamięci. Umożliwia on między innymi poruszanie obiektów na ekranie, rysowanie linii, itp. *Agnus* zawiera także COPPER (koprocessor video — odpowiedzialny za wyświetlanie i odświeżanie obrazu) oraz DMA (kanały, umożliwiające bezpośredni dostęp układów do pamięci).

DENISE jest układem pomocniczym. Wykonuje wiele funkcji video niskiego poziomu. Tłumaczy wszystkie dane o obrazie dostarczone przez COPPER i generuje na wyjściu RGB odpowiednie sygnały, które są pobierane przez monitor i „zamieniane” na obraz. Pomaga też przy obsłudze portów joysticka.

PAULA jest głównym układem zarządzającym portami wejścia/wyjścia. Obsługuje on operacje dyskowe, transmisje szeregowo, niektóre operacje wejścia/wyjścia portów joysticka, przerwania i jest odpowiedzialny przede wszystkim za odtwarzanie dźwięku.

Oprócz powyższych układów, Amiga posiada dwa układy we/wy typu CIA 8250. Układy te obsługują wiele operacji we/wy (porty: równoległy, joysticków, klawiatury), a także obsługują kontroler stacji dysków i uruchamiają filtr dźwiękowy.

Obsługa wszystkich układów odbywa się za pomocą rejestrów od adresu \$dff000 do \$dff1fe (dla układów Agnus, Paula, Denise) oraz \$bfd000 – \$bfd00 (dla układu CIA-A) i \$bfe001 – \$bfe01 (dla układu CIA-B). Zmiana zawartości tych rejestrów spowoduje odpowiednią reakcję systemu, natomiast ich odczyt może dać informację o jego aktualnym stanie.

Każdy rejestr układów specjalizowanych ma długość jednego słowa (16 bitów), za wyjątkiem rejestrów układów CIA, które mają długość 1 bajtu (8 bitów). Jednak często zdarza się, że rejestry układów specjalizowanych połączone są w określone pary umożliwiające zapisanie informacji większej od 16 bitów — czyli 32 bity (długie słowo).

Z reguły rejestry posiadają inne adresy dla zapisu i odczytu. Na przykład do kontroli przerwań używamy dwóch rejestrów: z jednego możemy odczytać które przerwania są dostępne, a drugiego używa się w celu zapisu, przy czym zapisując tam jaką wartość, modyfikacji ulega zawartość pierwszego z tych rejestrów. Z tego względu rejestry mogą być trzech rodzajów:

- tylko zapisywalne (W — ang. *write*),
- tylko odczytywalne (R — ang. *read*),
- zarówno zapisywalne, jak i odczytywalne (R/W).

Rejestry tylko do odczytu, zwykle na końcu swojej nazwy mają literkę "R", na przykład: DMACONR, INTENAR. Nigdy nie należy zmieniać zawartość rejestrów przeznaczonych tylko do odczytu i próbować odczytu rejestrów przeznaczonych tylko do zapisu. Na rejestrach od \$dff000 do \$dff1fe nie można stosować instrukcji typu BSET czy BCLR.

3.4 Kanały DMA

Kanały DMA (ang. *Direct Memory Acces*) służą do komunikacji układów specjalizowanych i procesora z pamięcią operacyjną. Kontrolują one przepływ danych, a także uwalniają procesor od uciążliwych operacji (np. wysyłania kolejnych próbek dźwięku do odtwarzania, czy kolejnych instrukcji coppera). Kanały DMA możemy podzielić na następujące grupy:

- DMA bitplanów,
- DMA coppera,
- DMA blittera,
- DMA spriteów,
- DMA dysku,
- DMA kanału dźwiękowego nr 3,
- DMA kanału dźwiękowego nr 2,
- DMA kanału dźwiękowego nr 1,
- DMA kanału dźwiękowego nr 0.

DMA kontrolujemy za pomocą rejestrów DMACON i DMACONR (ich adresy to odpowiednio: \$dff096, \$dff002). Ustawiając odpowiednie bity, możemy włączać i wyłączać poszczególne kanały. Szczegółowy opis tych rejestrów zamieszczony jest w osobnym rozdziale.

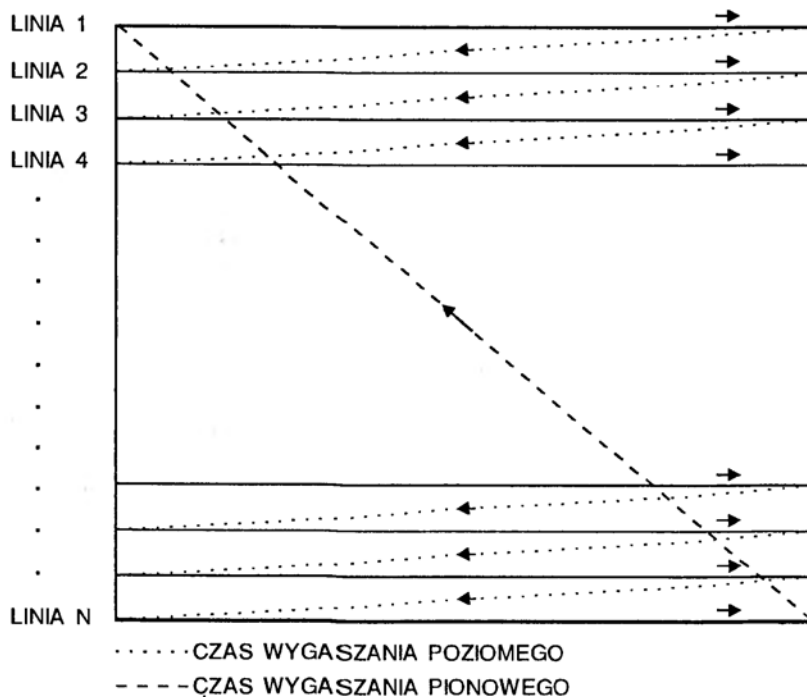
Przy pisaniu programów, trzeba pamiętać o otwieraniu dostępu systemu do odpowiednich kanałów, ponieważ czasami zdarza się, że po uruchomieniu programu, niczego nie ma na ekranie pomimo tego, że zdefiniowaliśmy jakiś obrazek.

Rozdział IV

Programowanie coppera

4.1 Charakterystyka coppera

Jak wcześniej wspomniałem, copper jest układem video. Wchodzi on w skład kości (chipu) Agnus. Tworzy on na podstawie zawartości określonych rejestrów, aktualną zawartość ekranu. Aby go lepiej wykorzystać, tworzy się tzw. copperlistę, czyli specjalny program, który jest wykonywany właśnie przez copper. W programie tym można operować tylko trzema rozkazami: MOVE, WAIT i SKIP. Wydaje się, że to bardzo mało, ale pozory mylą. Odpowiednie wykorzystanie tych rozkazów umożliwia uzyskanie niesamowitych efektów (łącznie z obrazkami obracanymi wektorowo).



Zanim przejdę do omawiania rozkazów coppera, muszę wyjaśnić zasadę tworzenia obrazu na ekranie monitora. W uproszczeniu, kineskop składa się z katody, układu skupiającego, układu odchyłającego i luminoforu. Elektrony emitowane przez katodę są skupiane w odpowiednio odchylaną wiązkę o bardzo małym przekroju. W efekcie promień elektronów uderza w luminofor, który świeci przez jakiś czas (ok 0.03 do 0.05 s). Wiązka ta omiata punkt po punkcie i linia po linii cały ekran. Gdy przejdzie od lewej do prawej jedną linię (pojedynczą linię nazywamy *linią rastra*) musi mieć czas na przejście do następnej linii. Okres ten nazywamy *wygaszaniem poziomym*. Gdy wiązka dojdzie do końca ekranu, musi powrócić do lewego górnego rogu — okres ten nazywamy *wygaszaniem pionowym*. Oko ludzkie nie jest w stanie tego zauważyć, ponieważ proces ten jest powtarzany 50 razy na sekundę. Czas, w którym wyświetlany jest jeden ekran nazywamy *ramką* (1/50 s), natomiast utworzony na ekranie obraz — *rastrum*.

Jednym z zadań coppera jest właśnie oczekiwanie z wykonaniem następnej instrukcji do czasu, w którym promień wizji znajdzie się w określonej pozycji.

4.2 MOVE, WAIT, SKIP — instrukcje coppera

Jako koprocesor, copper posiada własny zestaw instrukcji. Każda z nich składa się z długiego słowa (32 bitów). Wszystkie instrukcje zostaną wykonane w kolejności, w jakiej zostały zapisane.

4.2.1 Jak w assemblerze zapisać instrukcję coppera?

Gdy będziemy chcieli napisać program wykorzystujący copperlistę, możemy napotkać następujący problem: jak zapisać rozkazy copperlisty w naszym programie?

Jak wiadomo, każda instrukcja coppera jest 32-bitową liczbą. Gdy napiszemy w kodzie źródłowym:

```
$ffdfdfdf
```

to program assemblerujący poinformuje nas, że użyliśmy nielegalnego operatora (ang. *illegal operator*). Wobec tego, musimy określić, że to co chcemy umieścić w programie, jest zwykłą daną, która procesie assembleracji musi być umieszczona w pamięci. Do tego celu służy komenda assemblera **DC.X**, gdzie "X" jest rozmiarem danej (L, W, B). Wykorzystując tę komendę, dane umieszczone bezpośrednio za **DC.X** zostaną zapisane do pamięci w postaci binarnej. Jeśli więc zapiszemy:

```
DC.L    $ffdfdfdf
```

lub

```
DC.W    $ffdf,$dfdf
```

czy

```
DC.B    $ff,$df,$ff,$fe
```

to wszystko będzie prawidłowo. Jednak należy się tutaj małe ostrzeżenie. Nie wolno wpisywać danych większych niż zadeklarowane!

```
DC.B    $ffff          ;ŻLE !!!
DC.W    $b12a89        ;ŻLE !!!
```

```
DC.L    $1, -$2, $3           ; PRAWIDŁOWO
DC.W    $a112, $aa, $98bd     ; PRAWIDŁOWO
DC.B    $fe, $1, -$1          ; PRAWIDŁOWO
```

Oczywiście dane można także przedstawić jako liczby w systemach hexadecymalnym, decymalnym (dziesiętnym) i binarnym lub też jako tekst.

```
DC.W    10, $a, %00001010
DC.B    $a, %00001010, 10
DC.L    %00001010, 10, $a
DC.B    "To jest mój tekst!!!"
```

4.2.2 Instrukcja WAIT

Instrukcja WAIT zmusza copper do odczekania, aż pozycja promienia wizji będzie równa podanej w instrukcji lub większa.

Pierwsze słowo instrukcji zawiera pionowe i poziome współrzędne (koordynaty) pozycji promienia. Drugie słowo zawiera bity, które będą służyły do stworzenia *maski*, mówiącej które z bitów mają być porównywane.

Pierwsze słowo instrukcji

- bity 15 – 8 — pozycja pionowa promienia,
- bity 7 – 1 — pozycja pozioma promienia,
- bit 0 — zawsze ustawiony na 1.

Drugie słowo instrukcji

- bit 15 — bit blittera, normalnie ustawiony na 1 (jego działanie poznamy przy omawianiu blittera),
- bity 14 – 8 — które bity pozycji pionowej mają być odczytywane,
- bity 7 – 1 — które bity pozycji poziomej mają być odczytywane,
- bit 0 — zawsze ustawiony na 0.

Teraz przykład. Chcemy, aby copper zaczekał na linię 150 ignorując przy tym pozycję poziomą:

```
DC.L    $9601ff00
```

Kolejny przykład każe czekać copperowi na linię \$28:

```
DC.L    $2801fffe
```

Amiga może wyświetlić więcej linii, niż liczba, którą możemy podać w rozkazie coppera. Jeśli chcemy zaczekać na linię poniżej 255 (na przykład 276), należy zaczekać aż promień przejdzie do nowej części ekranu (do końca 255 linii) i od pozycji na którą chcemy czekać odjąć \$100.

W wyniku następnego przykładu, copper zaczeka na \$120 linię ekranu:

```
DC.L    $ffdfffff
DC.L    $2001fffe
```

4.2.3 Instrukcja MOVE

Działanie instrukcji MOVE polega na przestaniu słowa danych do określonego rejestru sprzętowego.

Pierwsze słowo instrukcji

- bity 15 – 9 — nie używane, powinny być skasowane,
- bity 8 – 1 — adres rejestru docelowego pomniejszony o \$dff000
(na przykład \$Dff180 – \$dff000 = \$180),
- bit 0 — zawsze skasowany.

Drugie słowo instrukcji

- bity 15 – 0 — słowo danej do przestania.

Oto przykład wpisujący adresy bitplanów do odpowiednich rejestrów coppera:

```
DC.L    $00e00002
DC.L    $00e20000
DC.L    $00e40002
DC.L    $00e62800
```

4.2.4 Instrukcja SKIP

Instrukcja SKIP zmusza copper do ominięcia następnej instrukcji jeżeli licznik promienia wizji jest większy lub równy wartości podanej w instrukcji.

Pierwsze słowo instrukcji

- bity 15 – 8 — pozycja pionowa promienia,
- bity 7 – 1 — pozycja pozioma promienia,
- bit 0 — zawsze ustawiony na 1.

Drugie słowo instrukcji

- bit 15 — bit blittera, ustawiać na 0 (przy używaniu blittera poznamy jego działanie)
- bity 14 – 8 — które bity pozycji pionowej mają być odczytywane
- bity 7 – 1 — które bity pozycji poziomej mają być odczytywane
- bit 0 — zawsze ustawiony na 1

Oto przykład instrukcji SKIP. Jeśli pionowa pozycja promienia będzie większa lub równa \$64, to program przeskoczy następną instrukcję — zignoruje pozycję poziomą.

```
DC.L    $6401ff01
```

4.3 Pozioma pozycja promienia wizji w instrukcjach coppera

Pozioma pozycja promienia może mieć wartość z zakresu od \$0 do \$e2. Najmłodszy bit przy porównywaniu nie jest używany, mamy więc $\$e2/2 = 113$ pozycji osiągalnych dla coppera. Odpowiada to czterem punktom w niskiej i ośmiu w wysokiej rozdzielczości. Wartości od \$f do \$35 zawarte są w okresie wygaszania poziomego.

4.4 Pionowa pozycja promienia wizji w instrukcjach coppera

Pionowa pozycja promienia może przyjąć wartość od 0 do 255 (\$00 – \$ff). Jednak w trybie PAL Amiga jest w stanie wyświetlić 312 linii, a w NTSC 262. Wobec tego, aby odwołać się do linii o numerze większym niż 255, należy zastosować następujący rozkaz WAIT:

```
DC.L    $ffdfaffe
```

Rozkaz ten powoduje oczekiwanie na moment, gdy promień wizji minie linię 255. Od tej chwili, aż do wygaszania pionowego, każda instrukcja będzie odnosić się do niższej części ekranu (czyli linii poniżej 255). Oto przykład:

```
DC.L    $ffdfaffe    ;oczekujemy na niższą część ekranu
DC.L    $1001fffe    ;czekamy na linie $10+$ff
```

4.5 Rejestry coppera

Copper posiada kilka rejestrów. Można je podzielić w następujący sposób:

- rejestry lokacji,
- komórki skoków,
- rejestr kontrolny.

4.5.1 Rejestry lokacji

Są to dwie pary rejestrów, określających położenie copperlist w pamięci. W celu ułatwienia pracy, copper posiada możliwość zdefiniowania dwóch copperlist.

Oto adresy tych rejestrów:

- COP1LCH — \$dff080 — 5 górnych bitów adresu copperlisty nr 1,
- COP1LCL — \$dff082 — 15 dolnych bitów adresu copperlisty nr 1,

\$0fff — biały,
 \$0666 — szary,
 \$0000 — czarny.

Poniżej podane są adresy rejestrów kolorów:

- COLOR00 — \$dff180
- COLOR01 — \$dff182
- COLOR02 — \$dff184
- ...
- ...
- ...
- COLOR31 — \$dff1be

Rejestr COLOR00 ma specjalne znaczenie — zawiera barwę tła. Jest ona wyświetlana w miejscach, gdy nie ma nic ponad nią.

4.7 Tworzenie copperlisty

Copperlistę zawsze kończymy specjalnym rozkazem WAIT — \$fffffffe.

Podczas tworzenia copperlisty należy pamiętać, aby wykonywać czynności w takiej kolejności, w jakiej promień wizji tworzy ekran. Oto przykład:

```
DC.L      $01800000      ; ustawiamy kolor tła na czarny
DC.L      $01000000      ; czyścimy rejestr $dff100
DC.L      $4c07ffff      ; czekamy na linię $4c
DC.L      $01800fff      ; zmieniamy kolor tła na biały
DC.L      $4e07ffff      ; czekamy na linię $4e
DC.L      $01800000      ; zmieniamy kolor tła na czarny
DC.L      $fffffffe      ; koniec copperlisty
```

Rezultatem będzie pasek białego koloru. Jednak wynik będzie zupełnie inny, gdy zrobimy tak:

```
DC.L      $01800000      ; ustawiamy kolor tła na czarny
DC.L      $01000000      ; czyścimy rejestr $dff100
DC.L      $4e07ffff      ; czekamy na linię $4e
DC.L      $01800000      ; zmieniamy kolor tła na czarny
DC.L      $4c07ffff      ; czekamy na linię $4c
DC.L      $01800fff      ; zmieniamy kolor tła na biały
DC.L      $fffffffe      ; koniec copperlisty
```

4.8 Kompletnie procedury uruchamiające copperlistę

Zanim zabierzemy się za pisanie procedury, musimy jeszcze poznać kilka rozkazów. Są nimi BTST, BSR, RTS, Bxx i JSR.

Rozkaz BTST (Test a bit — testowanie bitu)**Składnia:**

BTST #<dana>,<ea>

BTST Dn,<ea>

Atrybuty:

rozmiar: B, L

Działanie:

Instrukcja BTST testuje określony bit w operandzie przeznaczenia, i w zależności od jego wartości, ustawia bit Z rejestru statusowego. Numer bitu może być zawarty w rejestrze danych lub danej natychmiastowej.

Kody warunków:

X — nie zmieniany,

N — nie zmieniany,

Z — ustawiany, gdy testowany bit był zerowy,

V — nie zmieniany,

C — nie zmieniany.

Dozwolone tryby adresowania (dla statycznego numeru bitu):

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
tak	nie	tak	tak	tak	tak	tak
X.w	X.1	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	tak	tak	nie	nie	nie

Dozwolone tryby adresowania (dla dynamicznego numeru bitu):

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
tak	nie	tak	tak	tak	tak	tak
X.w	X.1	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	tak	tak	tak	nie	nie

Rozkaz BSR (Branch to subroutine — skok do podprogramu)**Składnia:**

BSR.S <etykieta>

BSR.W <etykieta>

Atrybuty:

rozmiar: B, W

Działanie:

Instrukcja BSR powoduje zapisanie na stosie systemowym adresu następnej instrukcji (wskazywanej przez PC). Następnie do rejestru PC (licznik programu) dodawane jest przesunięcie i od tak powstałego adresu wykonywane jest wykonywanie programu.

Kody warunków:

X — nie zmieniany,
 N — nie zmieniany,
 Z — nie zmieniany,
 V — nie zmieniany,
 C — nie zmieniany.

Rozkaz JSR (Jump to subroutine — skok do podprogramu)**Składnia:**

JSR <ea>

Atrybuty:

Bez rozmiaru

Działanie:

Instrukcja JSR wywołuje podprogram, którego adres określony jest adresem efektywnym. Powoduje także zapisanie na stosie systemowym adresu następnej instrukcji po JSR (wskazywanej przez PC).

Kody warunków:

X — nie zmieniany,
 N — nie zmieniany,
 Z — nie zmieniany,
 V — nie zmieniany,
 C — nie zmieniany.

Dozwolone tryby adresowania:

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
nie	nie	tak	nie	nie	tak	tak
X.w	X.l	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	tak	tak	nie	nie	nie

Przykład:

JSR \$60000

Wywołany zostanie podprogram znajdujący się pod adresem \$60000.

Rozkaz RTS (Return from subroutine — powrót z podprogramu)**Składnia:**

RTS

Atrybuty:

Brak

Działanie:

Instrukcja RTS odwraca działanie instrukcji BSR lub JSR. Licznik programu, zostaje załadowany długim słowem ze szczytu stosu. Powoduje to wykonywanie programu od instrukcji występującej zaraz po BSR lub JSR.

Kody warunków:

- X — nie zmieniany,
- N — nie zmieniany,
- Z — nie zmieniany,
- V — nie zmieniany,
- C — nie zmieniany.

Rozkaz Bxx (Branch conditionally — skok warunkowy)**Składnia:**

Bxx <etykieta>

Atrybuty:

rozmiar: B, W

Działanie:

Instrukcja Bxx jest instrukcją skoku warunkowego. Jeśli określony warunek zostanie spełniony, to mikroprocesor doda do PC adres, który będzie wskazywał następną instrukcję do wykonania. W przeciwnym wypadku nie nastąpi żadna zmiana.

Istnieją następujące skoki warunkowe:

- BCC — skok, gdy bit C jest wyzerowany,
- BCS — skok, gdy bit C jest ustawiony,
- BEQ — skok, gdy równy (bit Z jest ustawiony),
- BGE — skok, gdy większy lub równy (bity N i V=0 lub N i V=1),
- BGT — skok, gdy większy (bity N i V są ustawione, a bit Z wyzerowany lub bity N, V i Z są wyzerowane),
- BHI — skok, gdy wyższy (bity C i Z=0),

- BLE — skok, gdy mniejszy lub równy (bit Z=1 lub N=1 i V=0 lub N=0 i V=1),
- BLS — skok, gdy niższy lub taki sam (bit C lub Z=1),
- BLT — skok, gdy mniejszy (bity N=1 i V=0 lub N=0 i V=1),
- BMI — skok, gdy bit N jest ustawiony,
- BNE — skok, gdy różny (Z=0),
- BPL — skok, gdy większy od zera (N=0),
- BVC — skok, gdy V jest wyzerowany,
- BVS — skok, gdy V jest ustawiony,
- BRA — skok zawsze, niezależnie od sytuacji.

Kody warunków:

- X — nie zmieniany,
- N — nie zmieniany,
- Z — nie zmieniany,
- V — nie zmieniany,
- C — nie zmieniany.

Przykład:

```

SUB.L    D1,D0
BGE      Większe
...
Większe:
...
```

Jeśli zawartość rejestru D0 była większa bądź równa zawartości rejestru D1, to program będzie kontynuował działanie od etykiety Większe.

Teraz, możemy już napisać własny, prosty program, lecz zanim to zrobimy pozostaje nam do omówienia używanie etykiet w asemblerze.

Czym są etykiety? Są to wyszczególnione miejsca w programie (kodzie) źródłowym, które w procesie asemblacji zostają zamienione na odpowiednie adresy, liczby, itp. Mogą też być używane jako zmienne. Etykiety oznaczymy za pomocą prawie dowolnego ciągu liter i cyfr. Z reguły, powinny być one zakończone dwukropkiem (program staje się bardziej przejrzysty). Oto przykład:

```

;-----
Start:  MOVE.W  #$fff,$dff180    ;wpisz do rejestru koloru tła, barwę białą
        BTST.B  #6,$bfe001      ;czy lewy przycisk myszy przyciśnięty ???
        BNE     Start           ;jeśli nie, to idź do etykiety start
        RTS                      ;wróć z programu
;-----
```

Efektem tego programu będzie białe tło ekranu do czasu, gdy przyciśnięty zostanie lewy przycisk myszki. Gdybyśmy teraz zasemblowali ten programik od adresu \$20000, to etykiety Start, znajdującej się na samym początku programu przyporządkowany byłby adres \$20000. Instrukcja MOVE zaczyna się też od tego adresu.

Oto ten sam przykład, ale z innym wykorzystaniem etykiet. Część z nich potraktujemy jako zmienne.

```

;-----
Color00 =      $fff
Bit      =      6

Start:  MOVE.W  #Color00,$dff180;wpisz do rejestru koloru tła, zmienna COLOR00
        BTST.B  #Bit,$bfe001   ;czy lewy przycisk myszy przyciśnięty ???
        BNE     Start          ;jeśli nie, to idź do etykiety start
        RTS     ;wróć z programu
;-----

```

Jak widać, przed użyciem, zmienna musi zostać zadeklarowana.

Etykiety można używać również w ten sposób:

```

;-----
Start:  MOVE.W  Color00,$dff180 ;wpisz słowo spod etykiety COLOR00
        BTST.B  #6,$bfe001     ;czy lewy przycisk myszy przyciśnięty ???
        BNE     Start          ;jeśli nie, to idź do etykiety start
        RTS     ;wróć z programu

COLOR00:DC.W    $0fff
;-----

```

Efektom tego będzie wpisanie wartości spod adresu COLOR00 (w czasie asemblacji zostanie mu przy przypisany właściwy adres).

A oto pierwsza procedura inicjująca copperlistę:

```

Start:
        MOVE.L  #Copper_List,$dff080 ;wpisanie adresu copperlisty do
                                     ;rejestrów COP1LCH i COP1LCL
        MOVE.W  #0,$dff088           ;natychmiastowe uruchomienie copperlisty
Wait_Mouse_Button:
        BTST.B  #6,$bfe001          ;czy lewy przycisk myszy przyciśnięty?
        BNE     Wait_Mouse_Button    ;jeśli nie, to sprawdzaj dalej
        RTS     ;jeśli tak, to wyjdź z programu
;-----
Copper_List:
        DC.L    $01000000
        DC.L    $01800000           ;kolor tła-czarny
        DC.L    $4001ffff           ;czekamy na linię $40
        DC.L    $01800fff           ;ustawiamy kolor tła na biały
        DC.L    $4101ffff           ;czekamy na linię $41
        DC.L    $01800000           ;ustawiamy kolor tła na czarny
        DC.L    $fffffffe

```

W przykładzie tym należy zwrócić uwagę na rozkaz:

```
MOVE.W  #0,$dff088           ;natychmiastowe uruchomienie copperlisty
```

Co oznacza „natychmiastowe uruchomienie”? Gdybyśmy pominęli ten rozkaz, to copper wykonywałby starą copperlistę do czasu wygaszania pionowego. Dopiero po wygaszaniu pionowym pobrałby nowy adres copperlisty i zaczął ją wykonywać. Natychmiastowe uruchomienie ma na celu natychmiastowy „skok” coppera do nowej copperlisty.

Jednak teraz niektórzy z was pewnie są zdęgowani. Tyle tego wszystkiego, by wyświetlić jedną kreskę. Jednak nie tylko to można robić copperem. Oto kolejny przykład copperlisty:

```

;-----
Start:
    MOVE.L #Copper_List,$dff080 ;wpisanie adresu copperlisty do
                                ;rejestrów COP1LCH i COP1LCL
    MOVE.W #0,$dff088           ;natychmiastowe uruchomienie copperlisty
Wait_Mouse_Button:
    BTST.B #6,$bfe001           ;czy lewy przycisk myszy przyciśnięty?
    BNE    Wait_Mouse_Button    ;jeśli nie, to sprawdzaj dalej
    RTS                                     ;jeśli tak, to wyjdź z programu
;-----
Copper_List:
    DC.L    $01000000
    DC.L    $01800000           ;zmień kolor tła
    DC.L    $4001ffff           ;czekaj na linię $40
    DC.L    $01800fff           ;zmień kolor tła
    DC.L    $4101ffff           ;czekaj na linię $41
    DC.L    $01800eee           ;zmień kolor tła
    DC.L    $4201ffff           ;...
    DC.L    $01800ddd           ;...
    DC.L    $4301ffff           ;...
    DC.L    $01800ccc
    DC.L    $4401ffff
    DC.L    $01800bbb
    DC.L    $4501ffff
    DC.L    $01800aaa
    DC.L    $4601ffff
    DC.L    $01800999
    DC.L    $4701ffff
    DC.L    $01800888
    DC.L    $4801ffff
    DC.L    $01800777
    DC.L    $4901ffff
    DC.L    $01800666
    DC.L    $4a01ffff
    DC.L    $01800555
    DC.L    $4b01ffff
    DC.L    $01800444
    DC.L    $4c01ffff
    DC.L    $01800333
    DC.L    $4d01ffff
    DC.L    $01800222
    DC.L    $4e01ffff
    DC.L    $01800111
    DC.L    $4f01ffff
    DC.L    $01800000
    DC.L    $ffffff

```

Utworzyliśmy teraz w kolejnych liniach przejście kolorów od białego do czarnego. Copper zatrzymuje się na kolejnych liniach ekranu, by zmienić kolor tła.

Z kolei ta procedura demonstuje jedną z ciekawszych możliwości coppera.

```

;-----
Start:
    MOVE.L #Copper_List,$dff080
    MOVE.W #0,$dff088
Wait_Mouse_Button:
    BTST.B #6,$bfe001
    BNE    Wait_Mouse_Button
    RTS
;-----
Copper_List:
    DC.L    $01000000
    DC.L    $01800000           ;zmień kolor tła
    DC.L    $4041ffff           ;czekaj na linię
    DC.L    $01800fff           ;zmień kolor tła
    DC.L    $01800eee
    DC.L    $01800ddd
    DC.L    $01800ccc
    DC.L    $01800bbb

```

```
DC.L $01800aaa
DC.L $01800999
DC.L $01800888
DC.L $01800777
DC.L $01800666
DC.L $01800555
DC.L $01800444
DC.L $01800333
DC.L $01800222
DC.L $01800111
DC.L $01800000
DC.L $ffffffff
```

Copper po wykonaniu jednej instrukcji od razu wykonuje następną. Jednak wykonanie instrukcji zabiera trochę czasu, więc promień wizji zdąży zmienić swoje położenie. Tak więc następna instrukcja zmiany koloru tła odnosi się do innej pozycji ekranu.

4.9 Animacja

Animacja polega na okresowej zmianie kształtu lub pozycji danego obiektu. Zmiana odbywa się normalnie co każdą ramkę. Jednak im rzadziej będziemy wyświetlać (odświeżać) obraz, tym gorszej jakości będzie animacja.

Motorola, za pośrednictwem rejestrów sprzętowych, ma możliwość sprawdzania pozycji promienia co umożliwia zsynchronizowanie działania programu z czasem wyświetlania.

W poniższym przykładzie co 1/50 s będziemy zmieniać dane w copperliście, przez co uzyskamy wrażenie ruchu paska koloru.

```
;
Start:      MOVE.L    #Copper_List,$dff080    ; inicjuj copperlistę

Main_Loop:
BSR        Wait_Raster                        ; odczekaj na wygaszanie pion.
BSR        Bar_Sinus                          ; skocz do procedury animacji rastra
BTST.B     #6,$bfe001
BNE        Main_Loop
RTS

Wait_Raster:
MOVE.L     $dff004,d0                        ; odczekaj aż promień wizji
LSR.L      #8,D0                            ; osiągnie linię $12d
ANDI.W     #$1ff,d0
CMP.W      #$12d,d0
BNE        Wait_Raster
RTS

Bar_Sinus:
ADD.W      #1,Sinus_Position                ; dodaj 1 do wskaźnika pozycji
;w tablicy
ANDI.W     #$ff,sinus_position              ; wykonaj operację AND z $ff
MOVE.W     Sinus_Position,D0                 ; wpisz wskaźnik pozycji do D0
LEA        Sinus_Tab,A0                     ; załaduj adres początku tablicy do A0
MOVE.B     (A0,D0.w),D0                      ; zapisz element z tablicy do D0
MOVE.B     D0,Rast_Move1                    ; zapisz D0 jako pozycję
; pierwszego przerwania
ADD.W      #1,D0                             ; dodaj 1 do D0
MOVE.B     D0,Rast_Move2                    ; zapisz D0 jako pozycję
; drugiego przerwania
RTS
```

```

Copper_List:
    DC.L    $01000000
    DC.L    $01800000
Rast_Move1:
    DC.L    $4001ffff
    DC.L    $01800fff
Rast_Move2:
    DC.L    $4101ffff
    DC.L    $01800000
    DC.L    $fffffffe

Sinus_Position:
    DC.W    0                                ;wskaźnik danych w tablicy

Sinus_Tab:
                                ;tablica funkcji SINUS
    DC.B    $69,$6a,$6c,$6d,$6f,$71,$72,$74,$75,$77,$78,$7a,$7b,$7d,$7e,$80
    DC.B    $81,$83,$84,$85,$87,$88,$8a,$8b,$8c,$8d,$8f,$90,$91,$92,$94,$95
    DC.B    $96,$97,$98,$99,$9a,$9b,$9c,$9d,$9e,$9e,$9f,$a0,$a1,$a2,$a2,$a3
    DC.B    $a3,$a4,$a5,$a5,$a5,$a6,$a6,$a7,$a7,$a7,$a7,$a8,$a8,$a8,$a8,$a8
    DC.B    $a8,$a8,$a8,$a8,$a8,$a7,$a7,$a7,$a7,$a6,$a6,$a5,$a5,$a4,$a3
    DC.B    $a3,$a2,$a2,$a1,$a0,$9f,$9e,$9e,$9d,$9c,$9b,$9a,$99,$98,$97,$96
    DC.B    $95,$94,$92,$91,$90,$8f,$8d,$8c,$8b,$8a,$88,$87,$85,$84,$83,$81
    DC.B    $80,$7e,$7d,$7b,$7a,$78,$77,$75,$74,$72,$71,$6f,$6d,$6c,$6a,$69
    DC.B    $67,$66,$64,$63,$61,$5f,$5e,$5c,$5b,$59,$58,$56,$55,$53,$52,$50
    DC.B    $4f,$4d,$4c,$4b,$49,$48,$46,$45,$44,$43,$41,$40,$3f,$3e,$3c,$3b
    DC.B    $3a,$39,$38,$37,$36,$35,$34,$33,$32,$32,$31,$30,$2f,$2e,$2e,$2d
    DC.B    $2d,$2c,$2b,$2b,$2b,$2a,$2a,$29,$29,$29,$29,$28,$28,$28,$28,$28
    DC.B    $28,$28,$28,$28,$29,$29,$29,$29,$2a,$2a,$2b,$2b,$2b,$2c,$2d
    DC.B    $2d,$2e,$2e,$2f,$30,$31,$32,$32,$33,$34,$35,$36,$37,$38,$39,$3a
    DC.B    $3b,$3c,$3e,$3f,$40,$41,$43,$44,$45,$46,$48,$49,$4b,$4c,$4d,$4f
    DC.B    $50,$52,$53,$55,$56,$58,$59,$5b,$5c,$5e,$5f,$61,$63,$64,$66,$67

```

Zanim przejdziemy do szczegółowego omówienia tego przykładu, musimy poznać parę nowych rozkazów.

Rozkaz LEA (Load effective address — ładowanie adresu efektywnego)

Składnia:

LEA <ea>,An

Atrybuty:

rozmiar: L

Działanie:

Instrukcja LEA umieszcza w określonym rejestrze adresowym adres efektywny.

Kody warunków:

- X — nie zmieniany,
- N — nie zmieniany,
- Z — nie zmieniany,
- V — nie zmieniany,
- C — nie zmieniany.

Dozwolone tryby adresowania:

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
nie	nie	tak	nie	nie	tak	tak
X.w	X.1	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	tak	tak	nie	nie	nie

Przykład:

LEA \$20000, A0

W rejestrze A0 zostanie umieszczony adres \$20000.

Rozkaz MOVEA (Move address — przesłanie adresu)**Składnia:**

MOVEA <ea>, An

Atrybuty:

rozmiar: W, L

Działanie:

Instrukcja MOVEA przesyła adres określony przez operand źródłowy, do rejestru adresowego, przy czym zawartość operandu źródłowego pozostaje niezmieniona. Zmieniane są wszystkie 32 bity rejestru adresowego. Dana o długości słowa zostaje automatycznie rozszerzona do długiego słowa. Rozmiar operacji może być określony jako słowo lub długie słowo.

Kody warunków:

X — nie zmieniany,

N — nie zmieniany,

Z — nie zmieniany,

V — nie zmieniany,

C — nie zmieniany.

Dozwolone tryby adresowania (dla operandu źródłowego):

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
tak	tak	tak	tak	tak	tak	tak
X.w	X.1	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	tak	tak	tak	nie	nie

Przykład:

```
MOVEA.W D0, A0
```

Młodsze słowo z rejestru D0 zostanie skopiowane do młodszego słowa rejestru A0.
Starsze słowo rejestru A0 zostanie wyzerowane.

Rozkaz AND (AND logical — iloczyn logiczny)**Składnia:**

```
AND <ea>, Dn
```

```
AND Dn, <ea>
```

Atrybuty:

rozmiar: B, W, L

Działanie:

Instrukcja AND wykonuje operację iloczynu logicznego operandu źródłowego z operandem przeznaczenia. Wynik zostaje zapisany w miejscu przeznaczenia. Wielkość operacji może być określona jako bajt, słowo lub długie słowo.

Wynikiem iloczynu logicznego jest zawsze część wspólna. Jeśli więc mamy dwie cyfry binarne operandów, to wynik operacji będzie wynosił:

$$0 \text{ AND } 0 = 0$$

$$0 \text{ AND } 1 = 0$$

$$1 \text{ AND } 0 = 0$$

$$1 \text{ AND } 1 = 1$$

Oto przykład:

$$\begin{array}{r} \%01001101 \\ \text{AND } \%10111111 \\ \hline \%00001101 \end{array}$$

Kody warunków:

X — nie zmieniany,

N — ustawiany, gdy najbardziej znaczący bit wyniku jest równy jeden, w przeciwnym wypadku zerowany,

Z — ustawiany, gdy wynik jest równy zero, w przeciwnym wypadku kasowany,

V — zawsze zerowany,

C — zawsze zerowany.

Dozwolone tryby adresowania (adres efektywny określa operand źródłowy):

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
tak	nie	tak	tak	tak	tak	tak
X.w	X.l	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	tak	tak	tak	nie	nie

Dozwolone tryby adresowania (adres efektywny określa operand przeznaczenia):

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
nie	nie	tak	tak	tak	tak	tak
X.w	X.l	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	nie	nie	nie	nie	nie

Przykład:

AND.L D0,4(A2)

Na długim słowie zawartym w rejestrze D0 i danej spod adresu będącego sumą zawartości rejestru A2 i 4, zostanie wykonana operacja iloczynu logicznego.

Rozkaz ANDI (AND immediate — natychmiastowy iloczyn logiczny)

Składnia:

ANDI #dana,<ea>

Atrybuty:

rozmiar: B, W, L

Działanie:

Instrukcja ANDI wykonuje operację iloczynu logicznego natychmiastowego operandu źródłowego z operandem przeznaczenia. Wynik zostaje zapisany w miejscu przeznaczenia. Wielkość operacji może być określona jako bajt, słowo lub długie słowo.

Kody warunków:

X — nie zmieniany,

N — ustawiany, gdy najbardziej znaczący bit wyniku jest równy jeden, w przeciwnym wypadku zerowany,

Z — ustawiany, gdy wynik jest równy zero, w przeciwnym wypadku kasowany,

V — zawsze zerowany,

C — zawsze zerowany.

Dozwolone tryby adresowania:

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
tak	nie	tak	tak	tak	tak	tak
X.w	X.l	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	nie	nie	nie	tak	tak

Przykład:

```
DIVU    D1,D0
ANDI.L  #$0000ffff,D0
```

W tym przykładzie instrukcja ANDI służy do wykasowania reszty z dzielenia, która znajduje się w starszym słowie wyniku.

W praktyce ANDI stosuje się do:

- wykasowywania nieistotnych informacji w rejestrach,
- bezznakowego rozszerzania rozmiaru danych,
- ograniczenia wielkości parametrów do jakiejś potęgi dwójki.

Rozkaz CMP (Compare — porównanie)**Składnia:**

CMP (ea>,Dn

Atrybuty:

rozmiar: B, W, L

Działanie:

Działanie instrukcji CMP polega na porównaniu zawartości rejestru danych z operandem określonym adresem efektywnym. Porównanie to polega na odjęciu od rejestru danych operandu źródłowego i, w zależności od wyniku, odpowiedniego ustawienia kodów warunków. Żaden z operandów nie zostaje zmieniony. Rozmiar instrukcji jest dowolny.

Kody warunków:

- X — nie zmieniany,
- N — ustawiany gdy wynik jest ujemny, w przeciwnym wypadku zerowany,
- Z — ustawiany gdy wynik jest zerowy, w przeciwnym wypadku zerowany,
- V — ustawiany gdy wystąpił nadmiar przy odejmowaniu, w przeciwnym wypadku zerowany,
- C — ustawiany gdy wygenerowana została pożyczka, w przeciwnym wypadku zerowany.

Dozwolone tryby adresowania:

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
tak	tak	tak	tak	tak	tak	tak
X.w	X.1	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	tak	tak	tak	nie	nie

Przykład:

CMP.W D0,D1
BGE WiększeRówne

Jeśli zawartość rejestru D1 będzie większa od D0, to procesor skoczy do etykiety WiększeRówne (p. instrukcja Bxx).

W ostatnim przykładzie należało zwrócić uwagę na podprogram Bar_Sinus:, który pokazuje w jaki sposób można korzystać z tablic.

Program miał zdefiniowaną tablicę złożoną z 256 elementów, z których każdy miał długość jednego bajtu. Co każdą ramkę o 1 zmieniane było słowo pod adresem Sinus_Position:, które wskazywało odległość elementu w tablicy od jej początku. Aby uniknąć tego, że wskaźnik wskazywałby element za tablicą (czyli więcej od 255), wykonywana jest operacja Sinus_Position AND %11111111. Oznacza to, że jeśli Sinus_Position osiągnie wartość 256 (%100000000) to w wyniku tej operacji zostanie zmieniona na 0.

```

%100000000
AND %011111111
-----
%000000000

```

Element pobieramy z tablicy za pomocą odpowiedniego trybu adresowania.

Jednak sprawa byłaby inna, gdyby elementy w tablicy były rozmiaru słowa, czyli dwóch bajtów. Wskaźnik musiałby wtedy zmieniać się co 2 (wskazywałby słowo). Gdybyśmy zwiększali co 1, to zaraz zaszłaby sytuacja, w której wskazywałby połowę elementu w tablicy. Także operacja AND musiałaby się zmienić z 256 – 1 na 512 – 1 (ponieważ tablica jest dwukrotnie razy większa).

Rozdział V

Playfield

5.1 Wstęp

Amiga umożliwia wyświetlanie obrazu w różnych trybach, rozdzielczościach i ilościach kolorów. Zadaniem tego rozdziału jest nauka wyświetlania obrazu (zwanego playfieldem) i jego praktyczne wykorzystanie.

5.2 Podstawowe cechy playfieldu

Cały obraz na Amidze składa się z **pikseli**. Pikselem nazywamy pojedynczy punkt ekranu. Rząd pikseli układa się w linię. Ilość pikseli **widocznych** na ekranie w jednej linii, a także ilość linii zależy od **rozdzielczości**. Amiga posiada następujące tryby graficzne:

- LOWRES — 320×256 (piksele×linie) (PAL) lub 320×200 (NTSC),
- HIRES — 640×256,
- LOWRES+LACE — 320×512 (PAL) lub 320×400 (NTSC),
- HIRES+LACE — 640×512.

Oprócz tego możemy zdefiniować tzw. **okno wyświetlania**. Jest to miejsce na ekranie, w którym wyświetlamy jest obraz. Obrazek może być większy od okna wyświetlania. Okno wyświetlania definiuje ramki (czyli **border**) wokół ekranu. Jednak w Amidze ramka nie jest żadną stałą częścią ekranu. Definiując odpowiednio okno wyświetlania, możemy spowodować, że będzie ona niewidoczna na ekranie. Taki tryb nosi nazwę **overscan**.

Ważną rzeczą jest także ilość dostępnych kolorów w każdej z rozdzielczości.

LOWRES lub LOWRES+LACE	HIRES lub HIRES+LACE
2	2
4	4
8	8
16	16
32	
64 (EHB)	
4096 (HAM)	

Skróty w nawiasach określają specjalne tryby graficzne, w których kolory nie są od siebie w pełni niezależne.

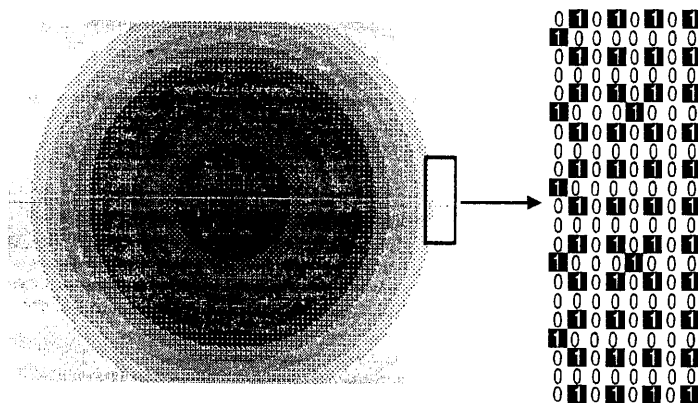
5.3 Zasady tworzenia prostego playfieldu

Aby wyświetlić obraz, należy zdefiniować następujące parametry:

- szerokość i wysokość playfieldu,
- okno wyświetlania,
- kolor każdego piksela w playfieldzie,
- rozdzielczość pionową i poziomą,
- sposób pobierania danych przez system.

5.3.1 Podstawowe rejestry — *bitplany i kolory*

Jak komputer wyświetla obrazki? Przypuśćmy, że mamy zwykły obrazek w dwóch kolorach, o wielkości 320 na 256 punktów (w najniższej rozdzielczości). Każdy punkt możemy potraktować jako bit. Jeśli w którymś miejscu jest ustawiony bit, to będzie tam zapalony punkt. To, co powstanie nazywamy *bitplanem* (płaszczyzną bitową).

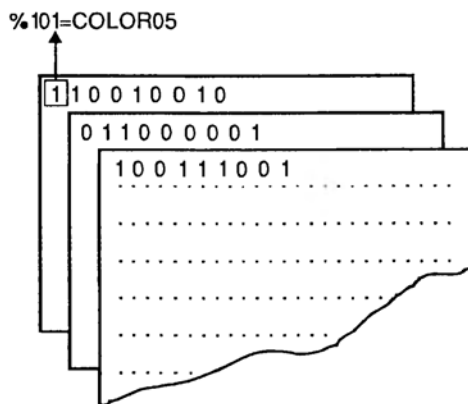


Przypuśćmy, że potrzebujemy teraz więcej niż dwa kolory. Wystarczy po prostu zwiększyć ilość bitplanów i „nałożyć” je na siebie. W każdym punkcie będzie występowała kombinacja bitów, która oznaczać będzie z którego rejestru koloru copper ma pobrać barwę i wyświetlić ją w tym miejscu, a do każdego rejestru koloru, możemy wpisać inną barwę:

- kombinacja %000, kolor 0, rejestr \$dff180, COLOR00,
- kombinacja %001, kolor 1, rejestr \$dff182, COLOR01,
- kombinacja %010, kolor 2, rejestr \$dff184, COLOR02,
- kombinacja %011, kolor 3, rejestr \$dff186, COLOR03,
- kombinacja %100, kolor 4, rejestr \$dff188, COLOR04,

- kombinacja %101, kolor 5, rejestr \$dff18a, COLOR05,
- kombinacja %110, kolor 6, rejestr \$dff18c, COLOR06,
- kombinacja %111, kolor 7, rejestr \$dff18e, COLOR07.

Liczbę kolorów dostępnych na ekranie określa liczba 2 podniesiona do potęgi ilości wyświetlanych bitplanów. Na przykład 3 bitplany włączone — 8 kolorów, bo $2^3=8$.



Położenie płaszczyzn bitowych w pamięci określamy w rejestrach BPLxPTH i BPLxPTL. BPLxPTH określa 5 górnych bitów, a BPLxPTL 16 dolnych bitów adresu początku płaszczyzny bitowej, należy pamiętać, aby adres był podzielny przez 2. Oto adresy tych rejestrów:

- \$dff0e0 — BPL1PTH
- \$dff0e2 — BPL1PTL
- \$dff0e4 — BPL2PTH
- \$dff0e6 — BPL2PTL
- \$dff0e8 — BPL3PTH
- \$dff0ea — BPL3PTL
- \$dff0ec — BPL4PTH
- \$dff0ee — BPL4PTL
- \$dff0f0 — BPL5PTH
- \$dff0f2 — BPL5PTL
- \$dff0f4 — BPL6PTH
- \$dff0f6 — BPL6PTL

Rejestry BPLCON0, BPLCON1, BPLCON2 o adresach:

- \$dff100 — BPLCON0
- \$dff102 — BPLCON1
- \$dff104 — BPLCON2

mają za zadanie kontrolę wyświetlanego obrazu. Przy ich pomocy definiujemy rozdzielczość ekranu, ilość kolorów, itp. Oto znaczenie poszczególnych bitów w tych rejestrach:

BPLCON0:

- bit 15 — ustawiony, włącza tryb HIRES (640 punktów w poziomie),
- bity 14 – 12 — kombinacja tych trzech bitów umożliwia włączenie danej liczby bitplanów:
- | | | |
|------|---------------|-------------------------|
| %000 | — 0 bitplanów | 1 kolor (tło) |
| %001 | — 1 bitplan | 2 kolory |
| %010 | — 2 bitplany | 4 kolory |
| %011 | — 3 bitplany | 8 kolorów |
| %100 | — 4 bitplany | 16 kolorów |
| %101 | — 5 bitplanów | 32 kolory |
| %110 | — 6 bitplanów | 64 kolory (EHB) lub HAM |
| %111 | — nie używane | |
- bit 11 — ustawiony włącza tryb HAM (ang. *Hold And Modify*), skasowany przy sześciu bitplanach oznacza tryb EHB (ang. *Extra Half Bright*),
- bit 10 — włącza tryb DOUBLE PLAYFIELD,
- bit 9 — używany w Amidze 1000 (włączenie kolorowego sygnału wideo),
- bit 8 — używany przy współpracy z genlockiem,
- bity 7 – 4 — nie używane,
- bit 3 — umożliwia współpracę z piórem świetlnym,
- bit 2 — włącza tryb INTERLACE,
- bit 1 — używany przy współpracy z genlockiem,
- bit 0 — nie używany.

BPLCON1:

- bity 15 – 8 — nie używane,
- bity 7 – 4 — określają przesunięcie parzystych bitplanów (od 0 do 15 punktów w prawo),
- bity 3 – 0 — określają przesunięcie nieparzystych bitplanów (od 0 do 15 punktów w prawo).

BPLCON2:

- bity 15 – 7 — nie używane,
- bit 6 — ustawiony oznacza, że bitplany parzyste będą wyświetlane przed bitplanami nieparzystymi (będą miały priorytet) — używany w trybie DUAL PLAYFIELD,
- bity 5 – 3 — określają priorytet PLAYFIELD-u 2 w stosunku do sprite'ów,
- bity 2 – 0 — określają priorytet PLAYFIELD-u 1 w stosunku do sprite'ów.

5.3.2 Przydzielanie pamięci na bitplany

Adres każdego bitplanu musi zostać umieszczony w rejestrach BPLxPT. Jednak często musimy przydzielić dla nich odpowiednie obszary pamięci. W tym celu korzystamy z następującego wzoru:

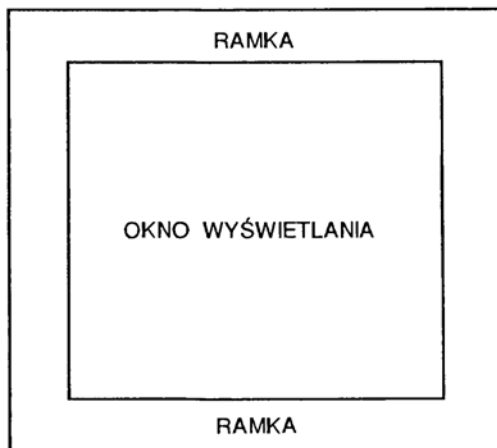
$$IPWP \cdot IL \cdot \frac{IB}{8}$$

gdzie:

- IPWP — liczba punktów w jednej linii poziomej obrazka,
- IL — ilość linii,
- IB — liczba bitplanów.

5.3.3 Definiowanie rozmiaru okna wyświetlania

Kiedy ustalimy już szerokość i wysokość playfieldu, musimy wyznaczyć *okno wyświetlania*. Jest to obszar ekranu, na którym wyświetlamy playfield lub jego część. Na pozostałej części ekranu, nic nie może już być wyświetlone. Okno mniejsze niż playfield, pozwala na wyświetlanie fragmentu jakiegoś playfieldu, jego animację (scrolling, czyli przesuwanie względem okna). Nikt też nie broni zdefiniowania okna większego niż playfield.

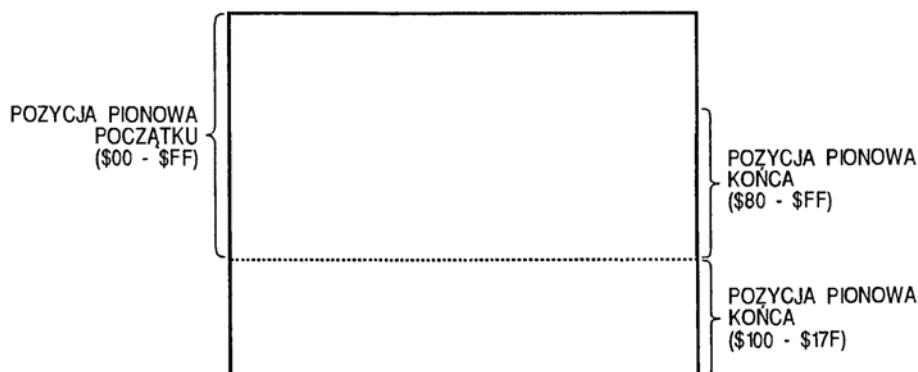


Rozmiar okna definiujesz przez określenie pozycji pionowych i poziomych początku i końca, oraz zapisanie wyników do rejestrów okna. Rozdzielczość startu i końca pionu to jedna linia, a rozdzielczość startu i końca poziomu, to jeden piksel niskiej rozdzielczości.

5.3.3.1 Ustawianie pozycji startowej i końcowej pionu.

Podzielmy cały ekran (łącznie z jego niewidoczną częścią) na \$17f części. Wtedy pozycja pionowa początku zawiera się w przedziale od \$00 do \$ff. Wartość \$2c zazwyczaj centruje obraz w większości monitorów. Natomiast pozycja końcowa jest podzielona na dwie części. Wartości z zakresu od \$00 do \$7f odpowiadają liniom od \$100 do \$17f, a z zakresu od \$80 do \$ff odpowiadają liniom od \$080 do \$0ff.

Jeśli zechcemy wyświetlić obrazek o wysokości 256 linii, ustawiamy pozycję początku na np. \$2c. W takim wypadku pozycja końca musiałaby znajdować się na linii \$12c. Wtedy należy pozycji nadać wartość \$2c, ponieważ pozycje końca od \$00 do \$7f odpowiadają liniom od \$100 do \$17f.



5.3.3.2 Ustawianie pozycji startowej i końcowej poziomu.

W wypadku ustawiania pozycji poziomu, dzielimy linię na \$1ff pikseli. I tak pozycje początku od \$00 do \$ff odpowiadają pikselom od \$000 do \$0ff, a pozycje końca od \$00 do \$ff odpowiadają pikselom od \$100 do \$1ff.

Jeśli mamy zamiar wyświetlić obrazek o szerokości 320 punktów, ustawiamy początek poziomu na \$81, a koniec na \$81+320 – \$100.



5.3.3.3 Zapisywanie rejestrów.

Jeśli określiliśmy już wartości początku i końca okna, to musimy je zapisać do rejestrów. Przypuśćmy, że wartości pionu wynoszą \$2c i \$2c, a wartości poziomu \$81 i \$c1. Łączymy wtedy wartości początku pionu z początkiem poziomu, oraz wartości końca pionu z końcem poziomu.

Otrzymujemy wtedy:

- Wartości początku — \$2c81
- Wartości końca — \$2cc1

I zapisujemy:

- W rejestrze DIWSTRT (adres \$dff08e) — pozycje początku,
- W rejestrze DIWSTOP (adres \$dff090) — pozycje końca.

Oto znaczenie poszczególnych bitów w tych rejestrach.

bit	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
	V7	V6	V5	V4	V3	V2	V1	V0	X7	X6	X5	X4	X3	X2	X1	X0

V0 do V7 — pozycja pionowa

X0 do X7 — pozycja pozioma

5.3.4 Sposób pobierania danych przez system

Po wyznaczeniu okna wyświetlania, należy zdefiniować lokację ekranową dla danych o obrazie, pobieranych z pamięci. Inaczej mówiąc, jest to wyznaczenie pozycji poziomej początkowej (zapisywanej do rejestru DDFSTRT) i końcowej (zapisywanej do DDFSTOP), między którymi (gdy je osiągnie wiązka elektronów) zacznie się pobieranie danych z pamięci i wyświetlanie ich na ekranie w tam samym miejscu. Rejestry w których zapisujemy te pozycje, mają rozdzielczość 4-ro pikselową. Oznacza to, że jednej pozycji odpowiadają 4 piksele.

Oto adresy rejestrów, w których zapisywane są informacje o lokacji obrazu:

- DDFSTRT — \$dff092 — pozioma pozycja początku pobierania danych,
- DDFSTOP — \$dff094 — pozioma pozycja końca pobierania danych.

Istnieje zależność pomiędzy szerokością okna wyświetlania, a pozycjami pobierania danych. Aby je obliczyć, korzystamy z następujących wzorów:

- dla wyłączonego trybu HIRES:

$$\text{DDFSTR} = \frac{\text{HS}}{2} - 8.5$$

$$\text{DDFSTOP} = \text{DDFSTR} + \frac{\text{ilośćpunktów}}{2} - 8$$

- dla włączonego trybu HIRES:

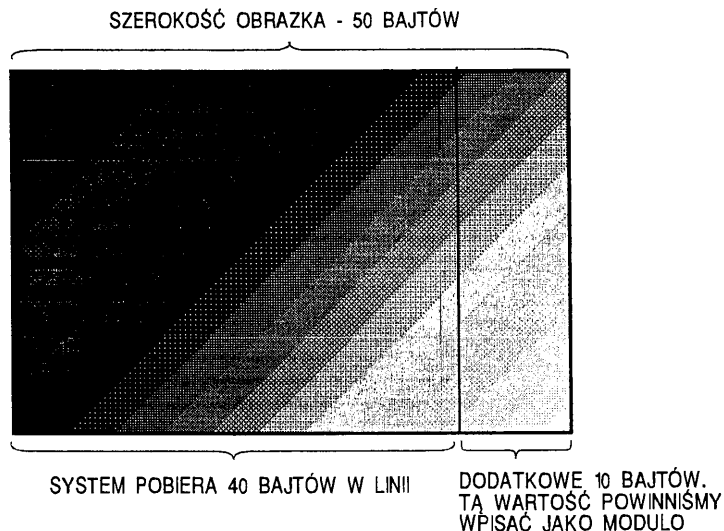
$$\text{DDFSTR} = \frac{\text{HS}}{2} - 4.5$$

$$\text{DDFSTOP} = \text{DDFSTR} + \frac{\text{ilośćpunktów}}{4} - 8$$

HS jest poziomą pozycją początku okna wyświetlania, czyli młodszym bajtem słowa zapisywanego do rejestru DIWSTRT.

Z pobieraniem danych przez system, łączy się jeszcze jedno pojęcie: MODULO.

Przypuśćmy, że mamy do wyświetlenia obrazek większy niż okno wyświetlania. System w każdej linii wyświetla kolejne 40 bajtów obrazka, a obrazek ma szerokość 50 bajtów. Wtedy po wyświetleniu pierwszej linii obrazka, w następnej linii ekranu zostanie wyświetlone końcowe 10 bajtów pierwszej linii obrazka i pierwsze 30 bajtów drugiej linii obrazka. Aby wszystko było dobrze, należałoby po wyświetleniu każdej linii ekranu dodawać 10 bajtów (czyli różnicę szerokości obrazka i ekranu). Takie zadanie ma właśnie modulo. Procesor po wyświetleniu każdej linii ekranu, dodaje wartości podane w rejestrach i zaczyna wyświetlanie następnej linii ekranu od adresu będącego sumą modulo i poprzedniego adresu. Modulo może być także ujemne. Oznacza to, że po wyświetleniu każdej linii ekranu, od wskaźników bitplanów modulo jest odejmowane (a nie jak wcześniej, dodawane).



Bitplany parzyste i nieparzyste mają osobne rejestry dla modulo. Dla bitplanów parzystych jest to BPL2MOD, a dla nieparzystych BPL1MOD. Rejestry te mają następujące adresy:

- BPL1MOD — \$dff108 — modulo bitplanów nieparzystych,
- BPL2MOD — \$dff10a — modulo bitplanów parzystych,

5.3.5 Rozdzielczości

Normalnie Amiga udostępnia rozdzielczość 320 punktów w poziomie (LOWRES). Można jednak włączyć tryb HIRES. Włączamy go ustawiając odpowiedni bit w rejestrze BPLCON0. Należy wtedy pamiętać, że rozdzielczość pozioma zostaje zwiększona z 320 do 640 punktów. Oznacza to, że powinniśmy pobierać 80 bajtów na linię.

Inaczej jest z rozdzielczością pionową. Mamy tutaj również do dyspozycji dwa tryby: zwykły (256 linii) i INTERLACE (512 linii). Jest to tryb z *przeplotem*. Oznacza to, że co 1/50 s na przemian wyświetlane są parzyste i nieparzyste linie playfieldu. Pozwala to na podwojenie rozdzielczości pionowej, a system pobiera co ramkę tyle samo danych o obrazie, co w trybach bez przeplotu. Niestety nie ma róży bez kolców. Skutkiem takiego postępowania jest dość dokuczliwe przy dłuższej pracy migotanie obrazu.

Aby włączyć ten tryb wyświetlania, trzeba ustawić bit LACE w rejestrze BPLCON0 i napisać dwie copperlisty, które uruchamiane są na przemian. Jedna wyświetla parzyste, druga nieparzyste linie obrazu. Odpowiedni przykład zostanie podany w części praktycznej tego rozdziału.

W trybie wyświetlania NTSC, liczba linii redukuje się do 200 lub 400 (w trybie intarlace).

5.3.6 Określenie zawartości bitplanów

Zawartość bitplanów możemy zdefiniować w różnoraki sposób. Wyświetlenie obrazka wymaga narysowania go na dowolnym edytorze graficznym i przekonwertowania (zmiany jego formatu) z IFF na RAW (czyli surową postać bitplanową). Można tego dokonać za pomocą jednego z wielu IFF CONVERTERów.

Można także w czasie rzeczywistym rysować w pamięci różne rzeczy (grafikę przestrzenną, zmieniające się wykresy funkcji, itp.) i wyświetlać ten fragment pamięci na ekranie.

5.3.7 Wielokrotne wyświetlanie playfieldu

Przy kolejnym wyświetlaniu tego samego playfieldu, należy zaadresować od nowa wskaźniki bitplanów. Są one zmieniane w wyniku wyświetlania kolejnych bajtów obrazu. Można tego dokonywać układając odpowiednią copperlistę.

5.3.8 Podsumowanie wiadomości o wyświetlaniu playfieldu

Oto podstawowe kroki definiowania najprostszego playfieldu:

- określ cechy playfieldu:
 - wielkość playfieldu,
 - rozdzielczość,
 - zdefiniuj bitplany,
- przydziel pamięć,
- zdefiniuj rozmiar okna wyświetlania,
- zdefiniuj pobieranie danych,
- ustaw MODULO.

5.3.9 Przykłady tworzenia prostego playfieldu

Oto przykład stworzenia prostego playfieldu. W najniższej rozdzielczości zostanie wyświetlony 16-sto kolorowy playfield o wielkości 320x256.

```

;-----
Start:      BSR      Init_Colors      ;ustawienie rejestrów kolorów
                                ;w copperliście
                                BSR      Init_Screens      ;wpisanie adresów bitplanów w
                                ;copperliście
Loop_Mouse: MOVE.L    #Copper,$dff080 ;uruchomienie copperlisty
            BTST     #6,$bfe001
            BNE      Loop_Mouse
            RTS

;-----
Init_Colors: LEA      Colors,A0        ;wartości spod etykiety Colors,
            LEA      Coplpalette,A1    ;traktujemy jako barwy i
            MOVE.L   #16-1,D7         ;wpisujemy do copperlisty
Col_Nt:     MOVE.W   (A0)+,2(A1)
            ADDA.L   #4,A1
            DBF      D7,Col_Nt
            RTS

;-----
Init_Screens: LEA      Planes,A0        ;wpisujemy adresy bitplanów
            MOVE.L   #Screen,D0        ;do instrukcji w copperliście
In_Scr:     MOVE.W   #4-1,D7
            MOVE.W   D0,6(A0)
            SWAP     D0
            MOVE.W   D0,2(A0)
            SWAP     D0
            ADD.L    #320*256/8,D0
            ADDA.L   #$8,a0
            DBF      D7,In_Scr
            RTS

;-----
Copper:     DC.L     $008e2c81,$00902cc1 ;definiujemy okno wyświetlania
            DC.L     $00920038,$009400d0 ;jak system ma pobierać dane
            DC.L     $00960020           ;sterujemy kanałami DMA
            DC.L     $01020000,$01040000 ;kasujemy BPLCON1 i BPLCON2
            DC.L     $01080000,$010a0000 ;modulo 0
Coplpalette: DC.L     $01800000,$01820000 ;zapisujemy rejestry kolorów
            DC.L     $01840000,$01860000
            DC.L     $01880000,$018a0000
            DC.L     $018c0000,$018e0000
            DC.L     $01900000,$01920000
            DC.L     $01940000,$01960000
            DC.L     $01980000,$019a0000
            DC.L     $019c0000,$019e0000
Planes:     DC.L     $00e00000,$00e20000 ;ustawiamy adresy bitplanów
            DC.L     $00e40000,$00e60000
            DC.L     $00e80000,$00ea0000
            DC.L     $00ec0000,$00ee0000
            DC.L     $01004000           ;włączamy 4 bitplany
            DC.L     $fffffffe

Colors:     DC.W     $0000,$0fff,$0eee,$0ddd,$0ccc,$0bbb,$0aaa,$0999
            DC.W     $0888,$0777,$0666,$0555,$0444,$0333,$0222,$0111
Screen:     DCB.B    $2800*4,$ff lub INCBIN 'DFO:PICTURE'

```

Po etykiecie **SCREEN** można użyć jednego z dwóch rozkazów. **DCB.B \$2800*4** w procesie asemblacji rezerwuje i wypełnia wartością po przecinku, podaną liczbę bajtów. W tym wypadku **DCB.X** nie jest instrukcją mikroprocesora, tylko daną dla programu asemblującego, kontrolującą sam proces asemblacji.

Natomiast **INCBIN 'DF0:PICTURE'** w procesie asemblacji dołącza do kodu wynikowego plik PICTURE. W naszym przypadku powinien to być 16-sto kolorowy obrazek, o wielkości 320x256, przekonwertowany na RAW.

Zanim przejdziemy do omówienia powyższego przykładu, musimy poznać znaczenie nowych użytych w nim rozkazów.

Rozkaz ADDA (Add address — dodawanie adresu)

Składnia:

ADDA <ea>,An

Atrybuty:

rozmiar: W, L

Działanie:

Instrukcja ADDA dodaje operand źródłowy, wskazywany odpowiednim trybem adresowania do określonego rejestru adresowego, który jest operandem przeznaczenia. Wynik zostaje zapisany w miejscu przeznaczenia. Rozmiar operacji określony jest jako słowo lub długie słowo. Instrukcję tę można zastąpić instrukcją LEA xxxx(An),An.

Kody warunków:

X — nie zmieniany,
N — nie zmieniany,
Z — nie zmieniany,
V — nie zmieniany,
C — nie zmieniany.

Dozwolone tryby adresowania:

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
tak	tak	tak	tak	tak	tak	tak
X.w	X.l	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	tak	tak	tak	nie	nie

Przykład:

ADDA.L #40,A1

Zawartość rejestru A1 zostaje zwiększona o 40.

Rozkaz SUBA (Subtract address — odejmowanie adresu)

Składnia:

SUBA <ea>,An

Atrybuty:

rozmiar: W, L

Działanie:

Instrukcja SUBA odejmuje operand źródłowy, wskazywany odpowiednim trybem adresowania od określonego rejestru adresowego, który jest operandem przeznaczenia. Wynik zostaje zapisany w miejscu przeznaczenia. Rozmiar operacji określony jest jako słowo lub długie słowo. Instrukcję tę można zastąpić instrukcją LEA xxxx(An),An.

Kody warunków:

X — nie zmieniany,
 N — nie zmieniany,
 Z — nie zmieniany,
 V — nie zmieniany,
 C — nie zmieniany.

Dozwolone tryby adresowania:

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
tak	tak	tak	tak	tak	tak	tak
X.w	X.l	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	tak	tak	tak	nie	nie

Przykład:

SUBA.W #360,A1

Zawartość młodszego słowa rejestru A1 zostanie zmniejszona o 360.

Rozkaz DBxx (Test condition, decrement and branch
— testowanie warunku, dekrementacja i skok)

Składnia:

DBxx Dn,<etykieta>

Atrybuty:

rozmiar: W

Działanie:

Instrukcja DBxx służy do sprzętowego tworzenia pętli w programie. Warunek xx jest podobny do warunków w skokach warunkowych. Podany warunek jest warunkiem zakończenia pętli. Jeśli nie jest on spełniony, to procesor skacze do określonego adresu. Licznik pętli znajduje się w młodszym słowie rejestru danych. Za każdym wykonaniem instrukcji DBxx jest on zmniejszany, aż do momentu, gdy osiągnie wartość -1. W tym momencie pętla zostaje zakończona.

Dostępne są następujące instrukcje:

- DBCC — zakończenie pętli, gdy bit C jest wyzerowany,
- DBCS — zakończenie pętli, gdy bit C jest ustawiony,
- DBEQ — zakończenie pętli, gdy równy (bit Z jest ustawiony),
- DBGE — zakończenie pętli, gdy większy lub równy (bity N i V=0 lub N i V=1),
- DBGT — zakończenie pętli, gdy większy (bity N i V są ustawione, a bit Z wyzerowany lub bity N, V i Z są wyzerowane),
- DBHI — zakończenie pętli, gdy wyższy (bity C i Z=0),
- DBLE — zakończenie pętli, gdy mniejszy lub równy (bit Z=1 lub N=1 i V=0 lub N=0 i V=1),
- DBLS — zakończenie pętli, gdy niższy lub taki sam (bit C lub Z=1),
- DBLT — zakończenie pętli, gdy mniejszy (bity N=1 i V=0 lub N=0 i V=1),
- DBMI — zakończenie pętli, gdy bit N jest ustawiony,
- DBNE — zakończenie pętli, gdy różny (Z=0),
- DBPL — zakończenie pętli, gdy większy od zera (N=0),
- DBVC — zakończenie pętli, gdy V jest wyzerowany,
- DBVS — zakończenie pętli, gdy V jest ustawiony,
- DBF — brak warunku zakończenia pętli, zostanie ona zakończona tylko gdy jej licznik będzie równy -1,
- DBT — natychmiastowe zakończenie pętli.

Kody warunków:

- X — nie zmieniany,
- N — nie zmieniany,
- Z — nie zmieniany,
- V — nie zmieniany,
- C — nie zmieniany.

Przykład:

```

                MOVE.W  #$2800-1,D0
Loop:          MOVE.B  #0,(A0)+
                DBF     D0,Loop

```

Zadaniem powyższego przykładu jest wyczyszczenie \$2800 bajtów pamięci spod adresu wskazywanego przez zawartość rejestru D0.

Rozkaz SWAP (Swap register halves — zamiana połówek rejestru)

Składnia:

SWAP Dn

Atrybuty:

rozmiar: W

Działanie:

Instrukcja SWAP zamienia miejscami słowa rejestru danych. Gdy w rejestrze będziemy mieli np. \$00062800, to po wykonaniu instrukcji SWAP, w rejestrze zostanie zapisana wartość \$28000006.

Kody warunków:

- X — nie zmieniany,
- N — ustawiany, gdy 31 bit rejestru danych jest równy 1, w przeciwnym wypadku zerowany,
- Z — ustawiany, gdy wszystkie 32 bity w rejestrze danych były równe 0, w przeciwnym wypadku zerowany,
- V — nie zmieniany,
- C — nie zmieniany.

Dozwolone tryby adresowania:

Dozwolony jest jedynie tryb bezpośredniego adresowania rejestru danych.

Powyższy przykład wyświetlenia prostego playfieldu wykorzystywał możliwość modyfikacji copperlisty przez mikroprocesor. Nie wiedzieliśmy, gdzie (pod jakim adresem) w procesie asemblacji, zostanie umieszczony obrazek, więc trzeba było napisać procedurę, która dopisywałaby brakujące słowa danych w instrukcjach coppera. Podobnie postąpiliśmy z paletą kolorów.

Czy nie mogliśmy jednak od razu założyć sobie, że obrazek znajdował się pod adresem, dajmy na to np. \$50000, i od razu zapisać to w instrukcjach coppera? Rzeczywiście, można tak było, ale byłoby to bardzo ryzykowne. Otóż system operacyjny Amigi dynamicznie przydziela pamięć dla wykonywanych programów i *nie* nigdy nie można z góry twierdzić, że dany obszar pamięci jest wolny. Jedynym wyjątkiem jest sytuacja, gdy całkowicie zablokujemy system operacyjny i przejmujemy kontrolę nad Amigą.

Aby zmienić procedurę na wyświetlającą obrazki w HIRES-ie, należy ustawić odpowiedni bit (w tym wypadku w copperliście zamieniamy DC.L \$01004000 na DC.L \$0100c000). Trzeba także odpowiednio zmodyfikować procedurę zapisującą adresy bitplanów do copperlisty i zmienić doczytywany obrazek.

5.3.10 Tryb interlace

Wyświetlenie obrazka w trybie interlace jest dość skomplikowane dla początkującego koder. Jak wcześniej wspomniałem, wymaga on zastosowania dwóch copperlist uruchamianych na przemian, z których jedna wyświetla linie parzyste, a druga linie nieparzyste obrazka. W tym celu włączamy bit LACE w BPLCON0 i w ustawiamy modulo tak, by wyświetlana była co druga linia obrazka. Aby jeden copper wyświetlał linie parzyste, a drugi nieparzyste, ustawiamy w jednym z nich adresy bitplanów o jedną linię niżej niż normalnie. Oto przykład:

```

;-----
Start:      MOVE.L  4.w,A6
            MOVE.L  156(A6),A1
            MOVE.L  38(A1),Old_Copper ;zapamiętujemy systemową copperlistę
            ESR     Init_Colors      ;wpisujemy kolory do copperlisty
            ESR     Init_Screens     ;inicjujemy adresy wyświetlania obrazu

```

```

BSR      Init_Coppers      ;modyfikujemy copperlisty tak, by
                           ;uruchamiały siebie nawzajem

Loop_Mouse:  MOVE.L  #Copper_1,$dff080
             BTST   #6,$bfe001
             BNE    Loop_Mouse
             MOVE.L  Old_Copper,$dff080
             MOVE.W  #0,$dff088
             RTS

;-----
Init_Colors: LEA     Colors,A0
             LEA     Cop1palette,A1
             MOVE.L  #15,D7
Col_Nt:      MOVE.W  (A0)+,2(A1)
             ADDA.L  #4,A1
             DBF     D7,Col_Nt
             RTS

;-----
Init_Screens: LEA     Planes_1,A0
              LEA     Planes_2,A1
              MOVE.L  #Screen,D0
              MOVE.W  #3,D7
In_Scr1:      MOVE.W  D0,6(A0)
              SWAP    D0
              MOVE.W  D0,2(A0)
              SWAP    D0
              ADD.L   #$28,d0
              MOVE.W  D0,6(A1)
              SWAP    D0
              MOVE.W  D0,2(A1)
              SWAP    D0
              ADD.L   #$2800-$28,d0
              ADDA.L  #$8,a0
              ADDA.L  #$8,a1
              DBF     D7,In_Scr1
              RTS

;-----
Init_Coppers: MOVE.L  #Copper_1,D0
              MOVE.W  D0,Cop_1_Run+6
              SWAP    D0
              MOVE.W  D0,Cop_1_Run+2
              MOVE.L  #Copper_2,D0
              MOVE.W  D0,Cop_2_Run+6
              SWAP    D0
              MOVE.W  D0,Cop_2_Run+2
              RTS

;-----
Copper_1:     DC.L    $008e2c81,$00902cc1
             DC.L    $00920038,$009400d0
             DC.L    $00960020
             DC.L    $01020000,$01040000
             DC.L    $01080028,$010a0028
Cop1palette:  DC.L    $01800000,$01820000
             DC.L    $01840000,$01860000
             DC.L    $01880000,$018a0000
             DC.L    $018c0000,$018e0000
             DC.L    $01900000,$01920000
             DC.L    $01940000,$01960000
             DC.L    $01980000,$019a0000
             DC.L    $019c0000,$019e0000
Planes_1:    DC.L    $00e00000,$00e20000
             DC.L    $00e40000,$00e60000
             DC.L    $00e80000,$00ea0000
             DC.L    $00ec0000,$00ee0000
             DC.L    $2c01ffff
             DC.L    $01004004
             DC.L    $ffdfffff
             DC.L    $2c01ffff
Cop_2_Run:   DC.L    $00800000
             DC.L    $00820000
             DC.L    $fffffffe

```

```

;-----
Copper_2:
Planes_2:    DC.L    $00e00000,$00e20000
              DC.L    $00e40000,$00e60000
              DC.L    $00e80000,$00ea0000
              DC.L    $00ec0000,$00ee0000
              DC.L    $2c01ffff
              DC.L    $01004004
              DC.L    $ffdfffff
              DC.L    $2c01ffff
Cop_1_Run:    DC.L    $00800000
              DC.L    $00820000
              DC.L    $fffffffe
;-----
Colors:       DC.W    $0000,$0fff,$0eee,$0ddd,$0ccc,$0bbb,$0aaa,$0999
              DC.W    $0888,$0777,$0666,$0555,$0444,$0333,$0222,$0fff
;-----
Screen:       DCB.B    $5000*4,$ff
;-----
Old_Copper:   DC.L    0
End:

```

W tym wypadku trzeba było zapamiętać standardową, systemową copperlistę, co czynią pierwsze trzy rozkazy. W jakim celu? Otóż copperlisty po zainicjowaniu przez mikroprocesor uruchamiają siebie nawzajem i po prostu wchodzą w pętlę. Aby ją przerwać, procesor musi wpisać adres starej copperlisty do rejestru lokacji i natychmiast rozkazać copperowi jej wykonywanie.

5.3.11 Rawblit

Jak dotychczas wyświetlaliśmy obrazki w których bitplany były ułożone w pamięci jeden za drugim (najpierw pierwszy bitplan, potem drugi, itd.). Jednak istnieje pewien sposób wyświetlania, który umożliwia szybsze wykonywanie operacji na grafice. Jest to RAWBLIT, czyli liniowe ułożenie bitplanów. Inaczej mówiąc, najpierw w pamięci jest ułożona pierwsza linia pierwszego bitplanu, potem pierwsza drugiego, pierwsza trzeciego, a następnie druga pierwszego, druga drugiego, itd. Wynika z tego że dla obrazka o szerokości 320 pikseli adres drugiego bitplanu powinien zaczynać się od adresu pierwszego + 40, itd. ...

Przypuśćmy, że chcemy wyświetlić obrazek składający się z 3 bitplanów (o szerokości 40 bajtów). Zapisujemy wtedy:

```

Bitplan_nr_1 = START
Bitplan_nr_2 = START+40
Bitplan_nr_3 = START+80

```

Jednak to nie wszystko. Gdybyśmy wyświetlili teraz nasz obrazek, ujrzelibyśmy barwną mozaikę, tylko trochę przypominającą nasz rysunek. Dzieje się tak dlatego, ponieważ komputer wyświetla:

	Bitplan_nr_1	Bitplan_nr_2	Bitplan_nr_3
Linia 1:	Linia_1_Bitplanu_1	Linia_1_Bitplanu_2	Linia_1_Bitplanu_3
Linia 2:	Linia_1_Bitplanu_2	Linia_1_Bitplanu_3	Linia_2_Bitplanu_1
Linia 3:	Linia_1_Bitplanu_3	Linia_2_Bitplanu_1	Linia_2_Bitplanu_2
Linia 4	...	...	...

Trzeba się więc jakoś pozbyć tych „nadprogramowych” linii. Z pomocą przychodzi nam tutaj modulo. Ustawiamy je tak, by pomijane zostały linie innych bitplanów, a wyświetlane tylko te potrzebne. Ustalamy je według wzoru:

$$(\text{ILP} - 1) \cdot \text{SZR}$$

gdzie:

ILP, to ilość bitplanów,

SZR, to szerokość obrazka w bajtach.

W naszym przypadku będzie to $(3 - 1) \cdot 40$, czyli 80.

Oto najprostszy przykład wyświetlania obrazka w trybie RAWBLIT. Należy pamiętać, by do pamięci wczytać obrazek przekonwertowany właśnie w taki sposób.

```

;-----
Start:      BSR      Init_Colors
            BSR      Init_Screens
            MOVE.L   #Copper,$dff080
Loop_Mouse: BTST     #6,$bfe001
            BNE      Loop_Mouse
            RTS

;-----
Init_Colors: LEA      Colors,A0
            LEA      Coplpalette,A1
            MOVE.L   #16-1,D7
Col_Nt:     MOVE.W   (A0)+,2(A1)
            ADDA.L   #4,A1
            DBF      D7,Col_Nt
            RTS

;-----
Init_Screens: LEA      Planes,A0
            MOVE.L   #Screen,D0
            MOVE.W   #4-1,D7
In_Scr:     MOVE.W   D0,6(A0)
            SWAP     D0
            MOVE.W   D0,2(A0)
            SWAP     D0
            ADD.L    #320/8,D0
            ADDA.L   #8,a0
            DBF      D7,In_Scr
            RTS

;-----
Copper:     DC.L     $008e2c81,$00902cc1
            DC.L     $00920038,$009400d0
            DC.L     $00960020
            DC.L     $01020000,$01040000
            DC.L     $01080078,$010a0078
Coplpalette: DC.L     $01800000,$01820000
            DC.L     $01840000,$01860000
            DC.L     $01880000,$018a0000
            DC.L     $018c0000,$018e0000
            DC.L     $01900000,$01920000
            DC.L     $01940000,$01960000
            DC.L     $01980000,$019a0000
            DC.L     $019c0000,$019e0000
Planes:     DC.L     $00e00000,$00e20000
            DC.L     $00e40000,$00e60000
            DC.L     $00e80000,$00ea0000
            DC.L     $00ec0000,$00ee0000
            DC.L     $2c01ffff
            DC.L     $01004000
            DC.L     $ffdfffff
            DC.L     $2c01ffff
            DC.L     $01000000
            DC.L     $fffffffe

```

Colors:	DC.W	\$0000,\$0fff,\$0eee,\$0ddd,\$0ccc,\$0bbb,\$0aaa,\$0999
	DC.W	\$0888,\$0777,\$0666,\$0555,\$0444,\$0333,\$0222,\$0111
Screen:	DCB.B	\$2800*4,\$ff lub INCBIN 'DF0:PICTURE'

5.3.12 Tryb EHB (Extra Half Bright)

Jak pewnie zauważyłeś, Amiga posiada 32 rejestry kolorów. Jednak przy włączonych 6 bitplanach i wyłączonym bicie HOMOD (w BPLCON0) możemy uzyskać 64 kolory. Jak wyświetlać takie obrazki?

Otóż „górna” część z tych 64 kolorów jest odbiciem drugiej połowy palety, ale ściemnionej o połowę. Postępujemy więc tak, jak ze zwykłym obrazkiem w 32 kolorach, tylko włączamy 6 bitplanów i wyłączamy bit HOMOD. Teraz tam, gdzie będzie zapalony bit na 6-stym bitplanie, barwa będzie pobierana z rejestru wskazywanego przez bity pierwszych 5-ciu bitplanów, ale zostanie ona ściemniona o połowę.

5.3.13 Tryb HAM (Hold And Modify)

Tryb HAM pozwala na wyświetlenie na ekranie całej palety kolorów Amigi. Jednak kolory te nie są od siebie niezależne.

Do wyświetlenia obrazu w HAM-ie, należy użyć 6 bitplanów, lecz kolor danego punktu jest określany zupełnie inaczej, niż zwykle. Bitplany 6 i 5 używane są do zmiany sposobu w jaki traktowane są wartości z bitplanów 1 do 4.

- Kombinacja %00. W tym wypadku kolor jest tworzony na podstawie bitplanów od 1 do 4. Barwa jest pobierana z odpowiedniego rejestru (od 0 do 15):
- Kombinacja %01 (pierwsza cyfra to bit z bitplanu nr 6). Kolor piksela bezpośrednio na lewo jest kopiowany i odpowiednio modyfikowany. Kombinacja bitów z 1 – 4, jest wstawiana jako dane składowe niebieskiej. Np. jeśli barwa punktu na lewo wynosi \$136, a kombinacja bitów w bitplanach 1 – 4 wynosi \$9, to kolor ostateczny wynosi:

$$\begin{array}{r}
 \text{RGB} \\
 \$136 \\
 \$009 \\
 \hline
 \$139
 \end{array}$$

- Kombinacja %10. Kolor piksela bezpośrednio na lewo jest kopiowany i odpowiednio modyfikowany. Kombinacja bitów z 1 – 4, jest wstawiana jako dane składowe czerwonej. Np. jeśli barwa punktu na lewo wynosi \$5fe, a kombinacja bitów w bitplanach 1 – 4 wynosi \$1, to kolor ostateczny wynosi:

$$\begin{array}{r}
 \text{RGB} \\
 \$5\text{fe} \\
 \$010 \\
 \hline
 \$51\text{e}
 \end{array}$$

- Kombinacja %11. Kolor piksela bezpośrednio na lewo jest kopiowany i odpowiednio modyfikowany. Kombinacja bitów z 1 – 4, jest wstawiana jako dane składowe zielonej. Np. jeśli barwa punktu na lewo wynosi \$dc6, a kombinacja bitów w bitplanach 1 – 4 wynosi \$f, to kolor ostateczny wynosi:

$$\begin{array}{r}
 \text{RGB} \\
 \$dc6 \\
 \$f00 \\
 \hline
 \$fc6
 \end{array}$$

Jeśli włączymy tylko 5 bitplanów, to wtedy wartość bitu z 6 bitplanu będzie zawsze równa 0.

Tryb HAM ustawiasz za pomocą zmiany następujących bitów w rejestrze BPLCON0:

- bit HOMOD na 1,
- bit DBLPF na 0,
- bit HIRES na 0.

Oczywiście należy pamiętać o ustawieniu odpowiedniej ilości bitplanów.

5.4 Wyświetlanie w trybie DUAL PLAYFIELD

Czasami potrzebne jest wykreowanie i wyświetlenie dwóch playfieldów na raz. Mają one wtedy niezależne palety kolorów, mogą być oddzielnie animowane, itp. Do tego właśnie służy specjalny tryb — DUAL PLAYFIELD:

Przypuśćmy, że chcesz wyświetlić grafikę przestrzenną, na przykład statek, na tle kosmosu. Wtedy na jednym playfieldzie możesz wyświetlić statek, a na drugim rysunek przestrzeni kosmicznej. Oto podstawowe różnice pomiędzy trybem DUAL PLAYFIELD (DPM), a zwykłym playfieldem:

- każdy playfield w DPM może mieć maksymalnie 8 kolorów (1,2 lub 3 bitplany),
- kolory każdego playfieldu są pobierane z innych rejestrów kolorów,
- każdy playfield posiada swój tzw. kolor przezroczysty (odpowiednik koloru tła).

5.4.1 Wyznaczanie bitplanów w trybie DUAL PLAYFIELD

Bitplany w trybie DPM są podzielone na dwie grupy: parzyste i nieparzyste. Bitplany nr 1, 3, 5 mogą być użyte w playfieldzie nr 1, a bitplany 2, 4, 6 w playfieldzie 2.

Przy włączonym trybie HIRES, liczba dostępnych kolorów dla jednego playfieldu zmniejsza się do czterech.

Ilość włączonych bitplanów	Playfield 1	Playfield 2
0	nic	nic
1	1	nic
2	1	2
3	1, 3	2
4	1, 3	2, 4
5	1, 3, 5	2, 4
6	1, 3, 5	2, 4, 6

5.4.2 Rejestry kolorów w trybie DUAL PLAYFIELD

W trybie DPM kolory nie są pobierane z kolejnych bitplanów, lecz z bitplanów przypadających na dany playfield. Oznacza to, że dla playfieldu nr 1 istotne są tylko bity pobrane z bitplanów 1, 3 i 5, a dla playfieldu nr 2 bity z bitplanów 2, 4, 6.

W wypadku, gdy kombinacja bitów będzie wynosić %000 (kolor przezroczysty), wtedy w tym miejscu zostanie wyświetlony drugi playfield, lub tło.

5.4.3 Priorytety playfieldów

Tryb DPM pozwala na określanie priorytetów wyświetlania playfieldów. Dzięki temu możemy swobodnie określić który z playfieldów ma być wyświetlany „wyżej” (nad drugim). Dokonujemy tego za pomocą rejestru BPLCON2:

- bit 6 włączony — Playfield 2 jest wyświetlany nad playfieldem 1,
- bit 6 wyłączony — Playfield 1 jest wyświetlany nad playfieldem 2.

5.4.4 Uruchomienie trybu DUAL PLAYFIELD

Aby uruchomić tryb DPM, należy ustawić 10-ty bit w rejestrze BPLCON0. Od tego momentu system automatycznie pogrupuje bitplany na parzyste i nieparzyste.

5.4.5 Podsumowanie wiadomości o trybie DUAL PLAYFIELD

Postępowanie przy tworzeniu playfieldów jest prawie identyczne, jak w wypadku tworzenia zwykłego playfieldu. Należy jednak pamiętać o:

- budowie playfieldów — playfield 1 składa się z bitplanów parzystych, a playfield 2 nieparzystych,
- rejestrach kolorów — kolory 0 – 7 są używane przez playfield 1, a 8 – 15 przez playfield 2,
- ustawieniu moduło — należy odpowiednio zapisać moduło dla playfieldu 1 i playfieldu 2,
- zdefiniowaniu priorytetów — jeśli chcesz, aby playfield 2 był wyświetlany nad playfieldem 1, ustaw bit PF2PR1 w BPLCON2,
- aktywizacji trybu — włącz bit DBLPF w BPLCON0.

5.5 Bitplany i okna wyświetlania wszystkich rozmiarów

Czasami potrzebujemy zdefiniować obraz większy niż okno wyświetlania lub też okno wyświetlania większe niż normalnie.

5.5.1 Bitplany większe od okna wyświetlania.

W wypadku bitplanów większych od okna wyświetlania, musimy określić następujące rzeczy:

- fragment obrazu, który chcemy wyświetlić,
- określić moduło.

Dla obrazka zdefiniowanego w pamięci, większego niż okno wyświetlania, musimy określić moduło tak, by pomijane były niewidoczne na ekranie części linii.

Oto przykład.

Mamy w pamięci zdefiniowany obrazek o szerokości 80 bajtów (640 punktów). Okno wyświetlania mamy zdefiniowane na 320 pikseli (system pobiera 40 bajtów na linię). W celu ustawienia moduło, korzysta się z prostego wzoru:

$$\text{Moduło} = (\text{Szerokość obrazka}) - (\text{Bajty na linię})$$

W naszym wypadku będzie to: $80 - 40 = 40$. Ustawiamy moduło na 40.

Gdy jednak zechcemy wyświetlić inną część tego samego obrazu, musimy zmienić tylko adresy początkowe bitplanów. Wyświetlmy obraz od 96 piksela. Dodajemy więc do adresu początkowego bitplanu 96/8, czyli 12 bajtów.

5.5.2 Maksymalny rozmiar okna wyświetlania

Maksymalny rozmiar okna jest ograniczony maksymalną ilością linii i kolumn możliwych do wyświetlenia. W pionie dane nie mogą być wyświetlane w obszarze wygaszania pionowego, czyli czasie, w którym wiązka elektronów wraca na początkową pozycję (do lewego górnego rogu ekranu). Oznacza to, że możemy wyświetlić obszar o maksymalnej wysokości 283 linii (w trybie INTERLACE — 567 linii).

W poziomie sytuacja jest dość podobna. Możemy tutaj wyświetlać obraz w obszarze, gdy DDFSTRT i DDFSTOP mają maksymalnie \$18 i \$d8. Przy wyłączonym trybie HIRES, daje to nam maksymalnie 50 bajtów na linię. W trybie HIRES włączonym, możemy pobrać maksymalnie 98 bajtów na linię.

	lores	hires
DDFSTRT (normalnie)	\$38	\$3c
DDFSTOP (normalnie)	\$d0	\$d4
DDFSTRT (max)	\$18	\$18
DDFSTOP (max)	\$d8	\$d8

5.6 Scrollowanie playfieldów

Scrollowanie oznacza przesuwanie się zawartości ekranu w jakąkolwiek stronę. W Amidze scrollowanie możemy podzielić na:

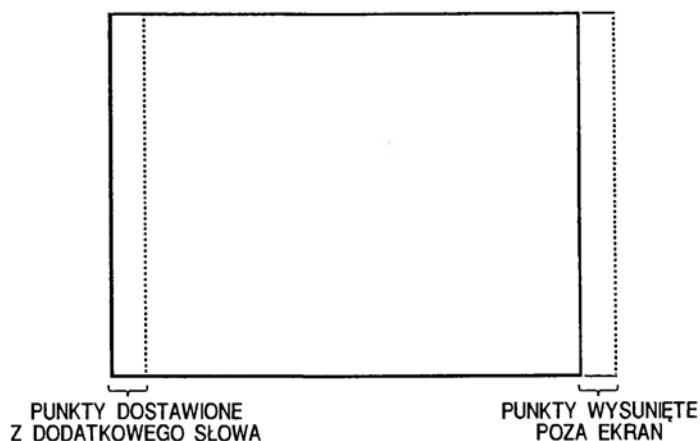
- sprzętowe,
- programowe.

W scrollowaniu programowym, przesuwamy zawartość ekranu za pomocą operacji programowych (np. mikroprocesorem, czy blitterem), natomiast w scrollowaniu sprzętowym, zmieniamy tylko adresy wyświetlania playfieldów. Oprócz tego, w wypadku scrollowania poziomego, zmieniamy zawartość rejestru BPLCON1. Aby jednak to zrobić, musimy zdefiniować ekran większy niż okno wyświetlania.

Oczywiście bez problemu możemy scrollować playfieldy w trybie DPM, każdy z osobna.

5.6.1 Scrollowanie poziome

W scrollowaniu poziomym, można zdefiniować ruch playfieldu z lewa na prawo lub odwrotnie. Z pomocą rejestru BPLCON1 możemy określić przesunięcie ekranu w prawo w granicach od 0 do 15 pikseli. Oznacza to jednak, że musimy pobierać jedno słowo więcej i modulo ustawiamy tak, by te dodatkowe słowo było pomijane. Pozornie bez sensu, prawda? Jednak wyobraźmy sobie taką sytuację: pobieramy 40 bajtów na linię i wyświetlamy 320 pikseli, przesuwamy ekran o 13 pikseli w prawo. Wtedy normalnie z lewej strony powstałaby dziura, ponieważ nie byłoby nic do wyświetlenia. Jednak pobierając jedno słowo więcej, od razu są dostawiane „zapasowe” piksele z tego dodatkowego słowa.



W trybie HIRES, scrollowanie odbywa się co 2 piksele. To znaczy, że jeśli ustawimy BPLCON1 na 12, to obraz zostanie wyświetlony z $12 * 2 = 24$ pikselowym przesunięciem.

5.6.1.1 Określanie pobierania danych w scrollowaniu poziomym

Jak wcześniej wspomniałem, musimy pobierać dodatkowe słowo w celu zapewnienia prawidłowego scrollowania. Robimy tak za pomocą rejestru DDFSTRT. Standardowo wynosi ono \$38. Odejmujemy więc od niego \$8 (czyli jedno słowo więcej do pobrania) i mamy nową wartość \$30, którą zapisujemy znowu do DDFSTRT. Nie zmieniamy DDFSTOP.

5.6.1.2 Określanie modulo w scrollowaniu poziomym

Konsekwencją pobierania dodatkowego słowa, jest ustawienie modulo tak, by wszystko grało. Oto przykład.

Mamy playfield szerokości 80 bajtów, a chcemy wyświetlać i scrollować 40 bajtów. Pobieramy więc dodatkowe słowo, wskutek czego co każdą linię ekranu adresy bitplanów są zwiększane co 42 bajty. Musimy więc pomijać różnicę $80 - 42$ czyli 38 bajtów, które nie są wyświetlane.

5.6.1.3 Określanie przesunięcia

Ilość bitów do przesunięcia ustawiamy w rejestrze BPLCON1. Bity 0 – 3 oznaczają przesunięcie bitplanów parzystych, a 4 – 7 bitplanów nieparzystych.

5.6.1.4 Dokładne omówienie zasady scrollowania

Dotychczas omawialiśmy tylko scrollowanie z prawa na lewo, w granicach od 0 do 15 pikseli. Przyszła pora na dokładne omówienie zagadnienia scrollowania.

Przypuśćmy, że chcemy scrollować z lewa na prawo. Wtedy na początku ustawiamy przesunięcie na 15 i co ramkę odejmujemy od niego 1. Gdy dojdzie do zera, to w następnej ramce od adresu bitplanów dodajemy jedno słowo i od nowa ustawiamy przesunięcie na 15. I tak w kółko.

W przypadku scrollowania z prawa na lewo, postępujemy dokładnie odwrotnie. Ustawiamy przesunięcie na 0 i co ramkę dodajemy do niego 1. Gdy dojdzie do 15, to w następnej ramce do adresu bitplanów odejmujemy słowo i od nowa ustawiamy przesunięcie na 0.

5.6.1.5 Przykład scrollowania poziomego

Poniżej przedstawiony jest przykład uniwersalnej procedury scrollowania ekranu w poziomie. Umożliwia ona scrollowanie ekranu w obydwie strony. Standardowo ustawione jest scrollowanie w prawo, lecz zmieniając rozkaz `ADD.W #1,Position_X`, na `SUB.W #1,Position_X`, możemy scrollować w lewo.

Do etykiety **SCREEN**: należy komendą **INCBIN** dograć obrazek w 2 kolorach, dowolnej szerokości, należy jednak pamiętać, by dostosować do niej moduło.

```

;
Start:      MOVE.L   #Copper,$dff080
Main_Loop:  BSR      Wait_Raster
            BSR      Scroll_Both
            ADD.W    #1,Position_X
            BTST     #6,$bfe001
            BNE      Main_Loop
            RTS

;-----
Wait_Raster: MOVE.L   $dff004,d0           ; odczekaj na wygaszanie pion.
            LSR.L    #8,D0
            ANDI.W   #$1ff,d0
            CMP.W    #$12d,d0
            BNE      Wait_Raster
            RTS

;-----
Scroll_Both: MOVE.W   Position_X,D0       ; wpisz pozycję do D0
            NEG      D0                   ; odwróć znak
            AND.W    #$f,d0              ; odczytaj przesunięcie dla
            MOVE.W   D0,D1                ; prześlij przesunięcie do D1
            ASL.W    #4,D1                ; przesun o 4 bity w lewo
            OR.W     D1,D0                 ; mamy przesunięcie dla bitplnów
            MOVE.W   D0,Scrolling+2      ; parzystych i nieparzystych
            MOVE.W   D0,Scrolling+2      ; zapisz przesunięcie do
            MOVE.W   D0,Scrolling+2      ; copperlisty
            MOVE.W   Position_X,D0       ; wpisz pozycję do D0
            ASR.W    #3,D0                ; oblicz przesunięcie w bajtach

```

```

                AND.L    #$ffff,d0                ; zignoruj najniższy bit (adres
                                                ; bitplanu musi być parzysty)
                MOVE.L   #Screen,D1
                ADD.W    D0,D1                    ; oblicz całkowity adres ekranu
                LEA      Planes,A0
                MOVE.W   D1,6(A0)
                SWAP     D1                        ; i wpisz go do copperlisty
                MOVE.W   D1,2(A0)
                RTS
;-----
Copper:        DC.L     $008e2c81,$00902cc1      ; definiujemy okno wyświetlania
                DC.L     $00920030,$009400d0      ; jak system ma pobierać dane
                DC.L     $00960020                ; sterujemy kanałami DMA
Scrolling:     DC.L     $01020000,$01040000      ; kasujemy BPLCON2
                DC.L     $01080038,$010a0038      ; modulo -2+40
Coplpalette:   DC.L     $01800000,$01820fff      ; zapisujemy rejestry kolorów
Planes:        DC.L     $00e00000,$00e20000      ; ustawiamy adres bitplanu
                DC.L     $2c01ffff
                DC.L     $01001000                ; włączamy 1 bitplan
                DC.L     $ffdfffff
                DC.L     $2c01ffff
                DC.L     $01000000                ; wyłączamy wyświetlanie bitplanów
                DC.L     $fffffffe
Screen:        Incbin   'Df0:Picture'
Position_X:    DC.W     0
;-----

```

Zanim omówimy zasadę działania tej procedury, musimy poznać garść nowych rozkazów.

Rozkaz ASR (*A*ritmetic *s*hift *r*ight — arytmetyczne przesunięcie w prawo)

Składnia:

```

ASR   Dn,Dm
ASR   #<dana>,Dn
ASR   <ea>

```

Atrybuty:

rozmiar: B, W, L

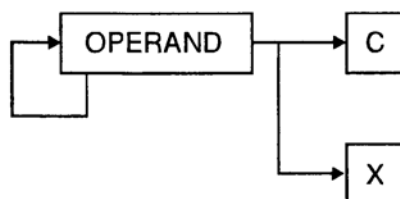
Działanie:

Działanie instrukcji ASR polega na przesunięciu arytmetycznym bitów operandu przeznaczenia w prawo. Najmłodszy bit jest kopiowany do znaczników C i X, natomiast najstarszy bit jest kopiowany do samego siebie, dzięki czemu zostaje zachowany znak wyniku. Istnieją trzy formy tej instrukcji:

- przesunięcie zawartości rejestru danych o liczbę zawartą w drugim rejestrze danych,
- przesunięcie zawartości rejestru danych o stałą liczbę zawartą w instrukcji (w tym wypadku możemy przesunąć tylko w zakresie od 1 do 8 bitów),
- przesunięcie zawartości słowa pod danym adresem, o jeden bit.

Rozmiar operacji może być określony jako bajt, słowo lub długie słowo. Dotyczy to tylko dwóch pierwszych trybów.

Rozkaz ten może zastąpić instrukcję dzielenia ze znakiem. Przesuwając daną o x, dzielimy ją przez 2^x .



Kody warunków:

- X — ustawiany zgodnie z ostatnim wysuniętym z operandu bitem,
- N — ustawiany, gdy najbardziej znaczący bit w wyniku operacji jest równy 1, w przeciwnym wypadku zerowany,
- Z — ustawiany, gdy wynik jest równy 0, w przeciwnym wypadku zerowany,
- V — zawsze zerowany,
- C — ustawiany zgodnie z ostatnim wysuniętym z operandu bitem.

Dozwolone tryby adresowania:

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
nie	nie	tak	tak	tak	tak	tak
X.w	X.l	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	nie	nie	nie	nie	nie

Przykład:

ASR (A0)

Słowo zawarte pod adresem określanym przez A0, zostanie przesunięte o 1 bit w prawo.

Rozkaz ASL (Arithmetic shift left — arytmetyczne przesunięcie w lewo)

Składnia:

ASL Dn,Dm
 ASL #<dana>,Dn
 ASL <ea>

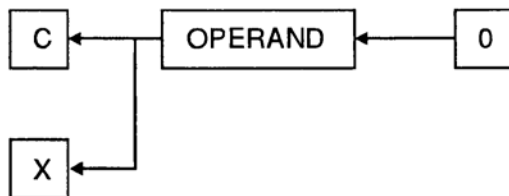
Atrybuty:

rozmiar: B, W, L

Działanie:

Działanie instrukcji ASL polega na przesunięciu arytmetycznym bitów operandu przeznaczenia w lewo. Najstarszy bit jest kopiowany do znaczników C i X, natomiast do najmłodszego bitu wpisywane jest 0. Formy tej instrukcji są identyczne jak instrukcji ASR.

Rozkaz ten może zastąpić instrukcję mnożenia ze znakiem. Przesuwając daną o x , mnożymy ją przez 2^x .



Kody warunków:

- X — ustawiany zgodnie z ostatnim wysuniętym z operandu bitem,
- N — ustawiany, gdy najbardziej znaczący bit w wyniku operacji jest równy 1, w przeciwnym wypadku zerowany,
- Z — ustawiany, gdy wynik jest równy 0, w przeciwnym wypadku zerowany,
- V — ustawiany wtedy, gdy najbardziej znaczący bit został zmieniony podczas wykonywania operacji przesunięcia, w przeciwnym wypadku zerowany,
- C — ustawiany zgodnie z ostatnim wysuniętym z operandu bitem.

Dozwolone tryby adresowania:

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
nie	nie	tak	tak	tak	tak	tak
X.w	X.1	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	nie	nie	nie	nie	nie

Przykład:

ASL, L #2, D0

W wyniku wykonania powyższej instrukcji, zawartość rejestru D0 zostanie pomnożona przez 4 (2^2).

Rozkaz LSR (Logical shift right — logiczne przesunięcie w prawo)

Składnia:

LSR Dn,Dm
 LSR #<dana>,Dn
 LSR <ea>

Atrybuty:

rozmiar: B, W, L

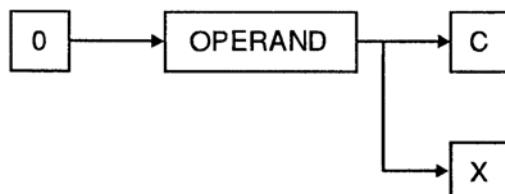
Działanie:

Instrukcja LSR przesuwają arytmetycznie bity operandu przeznaczenia w prawo. Najmłodszy bit idzie do znaczników X i C, a do najstarszego wpisywane jest 0. Istnieją trzy formy tej instrukcji.

- przesunięcie zawartości rejestru danych o liczbę zawartą w drugim rejestrze danych,
- przesunięcie zawartości rejestru danych o stałą liczbę zawartą w instrukcji (w tym wypadku możemy przesunąć tylko w zakresie od 1 do 8 bitów),
- przesunięcie zawartości słowa pod danym adresem, o jeden bit.

Rozmiar operacji może być określony jako bajt, słowo lub długie słowo. Dotyczy to tylko dwóch pierwszych trybów.

Rozkaz ten może zastąpić instrukcję dzielenia bez znaku. Przesuwając daną o x , dzielimy ją przez 2^x .

**Kody warunków:**

- X — ustawiany zgodnie z ostatnim wysuniętym z operandu bitem,
- N — ustawiany, gdy najbardziej znaczący bit w wyniku operacji jest równy 1, w przeciwnym wypadku zerowany,
- Z — ustawiany, gdy wynik jest równy 0, w przeciwnym wypadku zerowany,
- V — zawsze zerowany,
- C — ustawiany zgodnie z ostatnim wysuniętym z operandu bitem.

Dozwolone tryby adresowania:

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
nie	nie	tak	tak	tak	tak	tak
X.w	X.1	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	nie	nie	nie	nie	nie

Przykład:

LSR (A0)
LSR (A0)

Zawartość słowa wskazywanego przez zawartość rejestru A0 zostanie przesunięta o 2 bity w prawo, czyli podzielona przez 4.

Rozkaz LSL (Logical shift left — logiczne przesunięcie w lewo)**Składnia:**

LSL Dn,Dm
 LSL #<dana>,Dn
 LSL <ea>

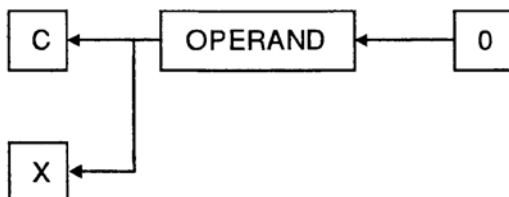
Atrybuty:

rozmiar: B, W, L

Działanie:

Instrukcja LSL przesuwaa arytmetycznie bity operandu przeznaczenia w lewo. Formy instrukcji takie same jak w LSR.

Rozkaz ten może zastąpić instrukcję mnożenia bez znaku. Przesuwając daną o x , mnożymy ją przez 2^x .

**Kody warunków:**

- X — ustawiany zgodnie z ostatnim wysuniętym z operandu bitem,
- N — ustawiany, gdy najbardziej znaczący bit w wyniku operacji jest równy 1, w przeciwnym wypadku zerowany,
- Z — ustawiany, gdy wynik jest równy 0, w przeciwnym wypadku zerowany,
- V — ustawiany wtedy, gdy najbardziej znaczący bit został zmieniony podczas wykonywania operacji przesunięcia, w przeciwnym wypadku zerowany,
- C — ustawiany zgodnie z ostatnim wysuniętym z operandu bitem.

Dozwolone tryby adresowania:

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
nie	nie	tak	tak	tak	tak	tak
X.w	X.1	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	nie	nie	nie	nie	nie

Przykład:

LSL (A0)

Zawartość słowa wskazywanego przez zawartość rejestru A0 zostanie przesunięta o 1 bit w lewo, czyli pomnożona przez 2.

Rozkaz ROR (Rotate right — obrót w prawo)**Składnia:**

ROR Dn,Dm
ROR #<dana>,Dn
ROR <ea>

Atrybuty:

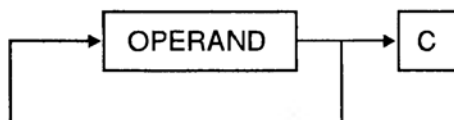
rozmiar: B, W, L

Działanie:

Instrukcja ROR dokonuje obrotu operandu przeznaczenia w prawo. Istnieją trzy formy tej instrukcji:

- obrót zawartości rejestru danych o liczbę zawartą w drugim rejestrze danych,
- obrót zawartości rejestru danych o stałą liczbę zawartą w instrukcji (w tym wypadku możemy przesunąć tylko w zakresie od 1 do 8 bitów),
- obrót zawartości słowa pod danym adresem, o jeden bit.

Rozmiar operacji może być określony jako bajt, słowo lub długie słowo. Dotyczy to tylko dwóch pierwszych trybów. Bity wysuwane z dolnej pozycji operandu idą do znacznika C i do górnej pozycji operandu.

**Kody warunków:**

- X — nie zmieniany,
N — ustawiany, gdy najbardziej znaczący bit w wyniku operacji jest równy 1, w przeciwnym wypadku zerowany,
Z — ustawiany, gdy wynik jest równy 0, w przeciwnym wypadku zerowany,
V — zawsze zerowany,
C — ustawiany zgodnie z ostatnim wysuniętym z operandu bitem.

Dozwolone tryby adresowania:

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
nie	nie	tak	tak	tak	tak	tak
X.w	X.l	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	nie	nie	nie	nie	nie

Rozkaz ROL (Rotate left — obrót w lewo)**Składnia:**

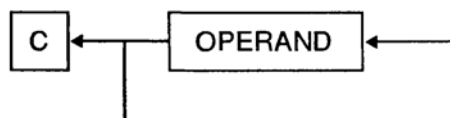
ROL Dn,Dm
 ROL #<dana>,Dn
 ROL <ea>

Atrybuty:

rozmiar: B, W, L

Działanie:

Instrukcja ROL dokonuje obrotu operandu przeznaczenia w lewo. Formy instrukcji takie same jak w ROR. Bity wysuwane z górnej pozycji operandu idą do znacznika C i do dolnej pozycji operandu.

**Kody warunków:**

- X — nie zmieniany,
- N — ustawiany, gdy najbardziej znaczący bit w wyniku operacji jest równy 1, w przeciwnym wypadku zerowany,
- Z — ustawiany, gdy wynik jest równy 0, w przeciwnym wypadku zerowany,
- V — ustawiany wtedy, gdy najbardziej znaczący bit został zmieniony podczas wykonywania operacji przesunięcia, w przeciwnym wypadku zerowany,
- C — ustawiany zgodnie z ostatnim wysuniętym z operandu bitem.

Dozwolone tryby adresowania:

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
nie	nie	tak	tak	tak	tak	tak
X.w	X.1	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	nie	nie	nie	nie	nie

Rozkaz ROXR (Rotate right with extend — obrót w prawo z rozszerzeniem)**Składnia:**

ROXR Dn,Dm
 ROXR #<dana>,Dn
 ROXR <ea>

Atrybuty:

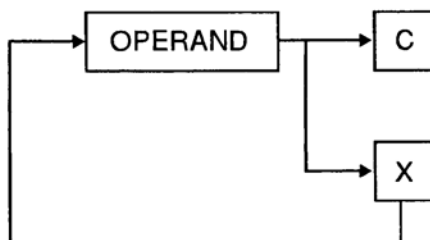
rozmiar: B, W, L

Działanie:

Instrukcja ROXR dokonuje obrotu operandu przeznaczenia w prawo. Istnieją trzy formy tej instrukcji.

- obrót zawartości rejestru danych o liczbę zawartą w drugim rejestrze danych,
- obrót zawartości rejestru danych o stałą liczbę zawartą w instrukcji (w tym wypadku możemy przesunąć tylko w zakresie od 1 do 8 bitów),
- obrót zawartości słowa pod danym adresem, o jeden bit.

Rozmiar operacji może być określony jako bajt, słowo lub długie słowo. Dotyczy to tylko dwóch pierwszych trybów. Bity wysuwane z dolnej pozycji operandu idą do znaczników X i C. Zawartość bitu C jest przesuwana do górnej pozycji operandu.

**Kody warunków:**

- X — ustawiany zgodnie z ostatnim wysuniętym z operandu bitem,
 N — ustawiany, gdy najbardziej znaczący bit w wyniku operacji jest równy 1, w przeciwnym wypadku zerowany,
 Z — ustawiany, gdy wynik jest równy 0. W przeciwnym wypadku zerowany,

- V — zawsze zerowany,
- C — ustawiany zgodnie z ostatnim wysuniętym z operandu bitem.

Dozwolone tryby adresowania:

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
nie	nie	tak	tak	tak	tak	tak
X.w	X.l	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	nie	nie	nie	nie	nie

Rozkaz ROXL (Rotate left with extend — obrót w lewo z rozszerzeniem)

Składnia:

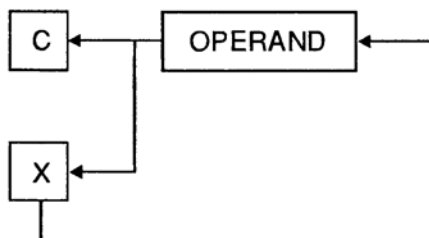
ROXL Dn,Dm
 ROXL #<dana>,Dn
 ROXL <ea>

Atrybuty:

rozmiar: B, W, L

Działanie:

Instrukcja ROXL dokonuje obrotu operandu przeznaczenia w lewo. Formy instrukcji takie same jak w ROXR. Bity wysuwane z górnej pozycji operandu idą do znaczników X i C. Zawartość bitu C jest przesuwana do dolnej pozycji operandu.



Kody warunków:

- X — ustawiany zgodnie z ostatnim wysuniętym z operandu bitem,
- N — ustawiany, gdy najbardziej znaczący bit w wyniku operacji jest równy 1, w przeciwnym wypadku zerowany,
- Z — ustawiany, gdy wynik jest równy 0, w przeciwnym wypadku zerowany,
- V — ustawiany wtedy, gdy najbardziej znaczący bit został zmieniony podczas wykonywania operacji przesunięcia, w przeciwnym wypadku zerowany,
- C — ustawiany zgodnie z ostatnim wysuniętym z operandu bitem.

Dozwolone tryby adresowania:

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
nie	nie	tak	tak	tak	tak	tak
X.w	X.l	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	nie	nie	nie	nie	nie

Rozkaz OR (Inclusive OR logical — suma logiczna)**Składnia:**

OR <ea>,Dn

OR Dn,<ea>

Atrybuty:

rozmiar: B, W, L

Działanie:

Instrukcja OR przeprowadza operację logicznego **lub** pomiędzy rejestrem danych i operandem określonym adresem efektywnym. Istnieją dwie formy tej instrukcji:

- operacja OR operandu określonego adresem efektywnym z rejestrem danych — wynik operacji zapisany będzie w rejestrze danych.
- operacja OR operandu określonego adresem efektywnym z rejestrem danych — wynik operacji zapisany będzie w miejscu określonym adresem efektywnym.

Na czym polega logiczne **lub**? Oto przykład:

$$\begin{array}{r}
 \%01001010 \\
 \text{OR } \%01010000 \\
 \hline
 \%01011010
 \end{array}$$

Można z niego wywnioskować, że:

$$0 \text{ OR } 0 = 0$$

$$0 \text{ OR } 1 = 1$$

$$1 \text{ OR } 0 = 1$$

$$1 \text{ OR } 1 = 1$$

Kody warunków:

X — nie zmieniany,

N — ustawiany, gdy najstarszy bit wyniku wynosi 1,

Z — ustawiany, gdy wynik wynosi 0,

V — zawsze zerowany,

C — zawsze zerowany.

Dozwolone tryby adresowania:

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
tak	nie	tak	tak	tak	tak	tak
X.w	X.1	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	nie	nie	nie	nie	nie

Rozkaz ORI (Inclusive OR immediate — natychmiastowa suma logiczna)

Składnia:

ORI <dana>, <ea>

Atrybuty:

rozmiar: B, W, L

Działanie:

Instrukcja ORI przeprowadza operację logicznego **lub** pomiędzy daną natychmiastową operandem przeznaczenia. Wynik operacji zapisany będzie w miejscu przeznaczenia.

Kody warunków:

X — nie zmieniany,

N — ustawiany, gdy najstarszy bit wyniku wynosi 1,

Z — ustawiany, gdy wynik wynosi 0,

V — zawsze zerowany,

C — zawsze zerowany.

Dozwolone tryby adresowania:

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
tak	nie	tak	tak	tak	tak	tak
X.w	X.1	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	nie	nie	nie	nie	nie

Rozkaz NEG (Negate — negacja)

Składnia:

NEG <ea>

Atrybuty:

rozmiar: B, W, L

Działanie:

Instrukcja NEG dokonuje operacji negowania na operandzie przeznaczenia. Wykorzystywana jest tutaj technika uzupełnienia do dwóch. Polega ono na odjęciu od samych jedynek określonego operandu i dodaniu do wyniku 1. W rezultacie z dodatniej liczby otrzymujemy ujemną i odwrotnie. Rozmiar operacji negowania może być określony jako bajt, słowo lub długie słowo.

Kody warunków:

- X — zerowany gdy wynik równy zero, w przeciwnym wypadku ustawiany,
- N — ustawiany, gdy wynik jest ujemny, w przeciwnym wypadku zerowany,
- Z — ustawiany, gdy wynik wynosi 0, w przeciwnym wypadku zerowany,
- V — ustawiany, gdy zgłoszony został nadmiar, w przeciwnym wypadku zerowany,
- C — zerowany gdy wynik równy zero, w przeciwnym wypadku ustawiany.

Dozwolone tryby adresowania:

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
tak	nie	tak	tak	tak	tak	tak
X.w	X.l	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	nie	nie	nie	nie	nie

Przykład:

```
MOVE.E #156,D0
NEG.W D0
```

Po wykonaniu tych dwóch instrukcji, w rejestrze D0 znajdzie się liczba -156.

Główną częścią procedury scrollowania w poziomie jest fragment oznaczony etykietą **Scroll_Both**. Na podstawie danej zawartej w **Position_X** oblicza odpowiedni adres bitplanów i przesunięcie ekranu za pomocą rejestru BPLCON1.

By procedura działała w obydwu kierunkach, trzeba było uogólnić sposób obliczenia przesunięcia zapisywanego w BPLCON1. Jeśli pozycja **X** wynosiła np. \$24, to program w celu obliczenia wartości przesunięcia, wykonywał następujące kroki:

```
MOVE.W Position_X,D0 ; w D0 $24
NEG D0 ; $db
AND.W #$f,d0 ; $0b
MOVE.W D0,D1 ; $0b do D1
ASL.W #4,D1 ; $b0 w D1
OR.W D1,D0 ; $b0 or $0b = $bb
```

Druga część procedury dzieliła przez 8 pozycję **X** i kasowała najmłodszy bit wyniku. Dzięki temu otrzymywaliśmy przesunięcie względem początkowego adresu ekranu w bajtach.

5.6.2 Scrollowanie pionowe

W przypadku scrollowania pionowego sprawa jest zupełnie prosta. Możemy playfield scrollować w górę lub w dół. Wystarczy co każdy okres wygaszania pionowego (ramkę) zwiększać lub zmniejszać o jedną linię adresy bitplanów.

Jeśli mamy duży obrazek o szerokości 40 bajtów i wysokości np. 1024 linii, to do wskaźników bitplanów dodajemy co ramkę 40 bajtów, dzięki czemu uzyskamy efekt scrollowania w dół. W każdym trybie, scrollowanie może odbywać się co 1 linię.

5.6.2.1 Przykład scrollowania pionowego

Poniższa procedura scrollowania playfieldu w górę i dół, działa na zasadzie zbliżonej do poprzedniego przykładu. Opiera się ona na pozycji zawartej w **Position_Y**. Procedura jest nieskomplikowana, więc jej analiza nie powinna sprawić Ci kłopotu. Żeby działała poprawnie, do etykiety **SCREEN:** należy komendą **INCBIN** dograć obrazek w 2 kolorach, dowolnej wysokości.

```

;-----
Start:      MOVE.L  #Copper,$dff080
Main_Loop:  BSR     Wait_Raster
            BSR     Scroll_Both
            ADD.W   #1,Position_Y
            BTST    #6,$bfe001
            BNE     Main_Loop
            RTS

;-----
Wait_Raster: MOVE.L  $dff004,d0          ; odczekaj na wygaszanie pion.
            LSR.L   #8,D0
            ANDI.W  #$1ff,d0
            CMP.W   #$12d,d0
            BNE     Wait_Raster
            RTS

;-----
Scroll_Both: MOVE.W  Position_Y,D0
            AND.L   #$ffff,d0
            Muls    #40,D0
            MOVE.L  #Screen,D1
            ADD.W   D0,D1
            LEA     Planes,A0
            MOVE.W  D1,6(A0)
            SWAP    D1
            MOVE.W  D1,2(A0)
            RTS

;-----
Copper:     DC.L    $008e2c81,$00902cc1 ; definiujemy okno wyświetlania
            DC.L    $00920038,$009400d0 ; jak system ma pobierać dane
            DC.L    $00960020             ; sterujemy kanałami DMA
Scrolling:   DC.L    $01020000,$01040000 ; kasujemy BPLCON2
            DC.L    $01080000,$010a0000 ; modulo 0
Copipalette: DC.L    $01800000,$01820fff ; zapisujemy rejestry kolorów
Planes:     DC.L    $00e00000,$00e20000 ; ustawiamy adres bitplanu
            DC.L    $2c01ffff
            DC.L    $01001000             ; włączamy 1 bitplan
            DC.L    $ffdfffff
            DC.L    $2c01ffff
            DC.L    $01000000             ; wyłączamy wyświetlanie bitplanów
            DC.L    $fffffffe
Screen:      Incbin  'Df0:Picture'
Position_Y:  DC.W    0

```


Rozdział VI

Blitter

6.1 Wprowadzenie

Wcześniej wspominałem, że blitter jest układem (a tak naprawdę częścią kości *Agnus*) umożliwiającym kopiowanie prostokątnych obszarów pamięci, z równoczesnym wykonywaniem na nich operacji logicznych. Potrafi on również wypełniać płaszczyzny, czy też rysować linie (z prędkością prawie 1 mln pikseli na sekundę). Umożliwia on operowanie na 4 MB pamięci w ciągu sekundy. Jest to jednak prędkość teoretyczna, ponieważ np. przeciążenie kanałów DMA bardzo zwalnia jego pracę. DMA blittera ma dostęp tylko do pamięci typu CHIP.

Podstawą pracy blittera są trzy kanały źródłowe (oznaczone A, B i C). Na podstawie zadanych parametrów, odpowiednio łączy on pobrane dane i wysyła jednym kanałem docelowym (oznaczonym D).

6.2 Określanie adresów kanałów i typu operacji.

Adresy, skąd mają być pobierane dane dla kanałów źródłowych wpisujemy do określonych rejestrów (oczywiście wpisujemy długie słowo):

- BLTAPT — adres rejestru \$dff050
- BLTBPT — adres rejestru \$dff048
- BLTCPT — adres rejestru \$dff04c

A oto rejestr, do którego wpisujemy adres kanału przeznaczenia:

- BLTDPT — adres rejestru \$dff054

Każdą operację blittera możemy zapisać w postaci równania i z jego pomocą obliczyć pewną wartość liczbową. Właśnie ta liczba, po wpisaniu jej do określonego rejestru, dokładnie opisze blitterowi żadaną operację. Jednak przypomnijmy sobie najpierw tabelki operacji logicznych *AND*, *OR* oraz przykład operacji negacji.

$$0 \text{ OR } 0 = 0$$

$$0 \text{ AND } 0 = 0$$

$$0 \text{ OR } 1 = 1$$

$$0 \text{ AND } 1 = 0$$

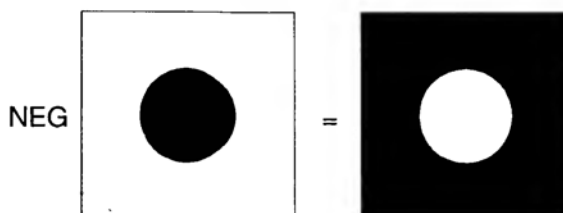
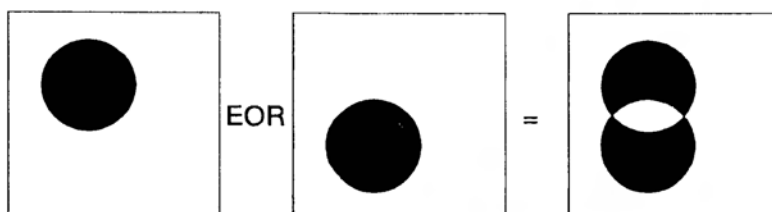
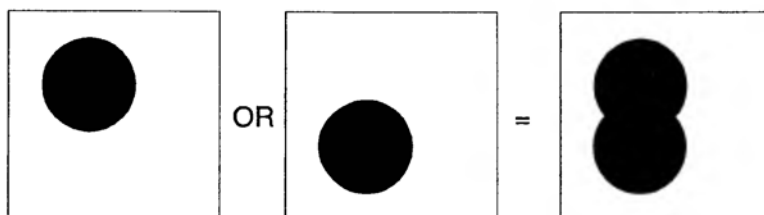
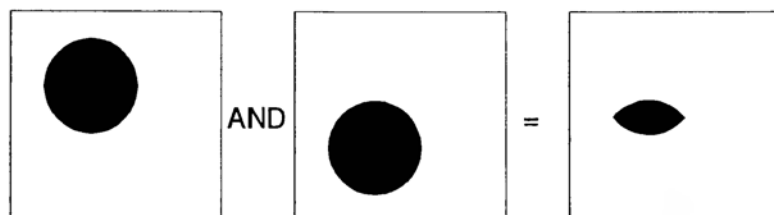
$$1 \text{ OR } 0 = 1$$

$$1 \text{ AND } 0 = 0$$

$$1 \text{ OR } 1 = 1$$

$$1 \text{ AND } 1 = 1$$

negacja: $\overline{\%0010011110101001} = \%1101100001010110$



Rejestrami odpowiadającymi za tryby pracy blittera są BLTCON0 (\$dff040) i BLTCON1 (\$dff042). Oto ich skrótowy opis.

BLTCON0:

bity 15 – 12 — ASH3 – ASH0 — określają przesunięcie portu A od 0 do 15 pikseli,

bit 11 — USEA — czy używany jest port A,

bit 10 — USEB — czy używany jest port B,

bit 9 — USEC — czy używany jest port C,
 bit 8 — USED — czy używany jest port D,
 bity 7 – 0 — LF7 – LF0 — określają rodzaj operacji.

BLTCON1:

bity 15 – 12 — BSH3 – BSH0 — określają przesunięcie portu B od 0 do 15 pikseli,
 bit 11 – 05 — nie używane,
 bit 4 — USEB — bit Exclusive Fill,
 bit 3 — IFE — bit Inclusive Fill,
 bit 2 — FCI — bit Fill Carry In,
 bit 1 — DESC — tryb DESCENDING,
 bit 0 — LINE — bit odpowiadający za tryb kreślenia linii.

Jak widać mamy 8 bitów określających rodzaj operacji logicznej, więc do dyspozycji mamy ich 2^8 , czyli 256. Każdy bit LFX ma przyporządkowaną kombinację bitów zwaną *mintermem*. Określa on, jaka kombinacja bitów z trzech źródeł powinna zostać dokonana.

LF7 = ABC	LF6 = ABc	LF5 = AbC	LF4 = Abc
LF3 = aBC	LF2 = aBc	LF1 = abC	LF0 = abc

Duże litery oznaczają odpowiednie kanały źródłowe, a małe to te same kanały ale z zanegowanymi danymi. Weźmy teraz minterm AbC. Oznacza to, że na trzech danych ze źródeł A, B i C, dokonana zostanie operacja:

A AND (NEG B) AND C

Z kolei ABC oznacza operację:

A AND B AND C

Oto przykład:

Minterm ABC

A AND B AND C

dana z kanału A	%0100010100111010
dana z kanału B	%1111011010101000
dana z kanału C	%1100000100101010
dana dla kanału B	%0100000000101000

Jeśli chcemy jednak uzyskać ciekawsze kombinacje, to znacznie wygodniejsze będzie układanie równań. Oto przykład: blitter ma za zadanie tylko kopiować jakiś obszar pamięci. Więc dana z kanału A powinna iść od razu do kanału D, czyli $D = A$. Ale musimy teraz uwzględnić pozostałe, nieużywane kanały w taki sposób, by żadna kombinacja danych pobrana z nich nie zniekształciła wyniku operacji. Ustalamy więc w równaniu, że ich wartości będą cały czas wynosić 1. Ale jak to zrobić?

Wartość w operacji OR z tą samą wartością, ale zanegowaną, da w wyniku same jedynki. Czyli operację np. dla portu B układamy tak: $B + b$ (czyli B or (neg B)). Ustalamy więc do końca równanie:

$$D = A(B+b)(C+c)$$

W sumie otrzymujemy: $D = A$ and 1 and 1. Czyli wszystko gra. Wystarczy teraz wymnożyć nawiasy:

$$D = A(BC+Bb+bC+bc)$$

I ostatecznie otrzymujemy:

$$D = ABC+ABc+AbC+Abc$$

Wystarczy teraz tylko popatrzeć, w jaki sposób mamy ustawić bity mintermów w BLTCON0. W naszym wypadku będą to LF7, LF6, LF5, LF4. Na razie nic nie przesuwamy i używamy tylko kanałów A i D, więc ostateczna zawartość rejestru BLTCON0 wynosi %0000100111110000 (\$09f0).

Teraz następny przykład. Chcemy wykonać operację $D = AB + aC$. Dopelniamy i otrzymujemy:

$$D = AB(C+c)+aC(B+b)$$

czyli:

$$D = ABC+ABc+aBC+abC$$

Ustawiamy teraz odpowiednie bity i mamy kombinację %11001010 (\$ca). W wypadku, gdybyśmy otrzymali kilka takich samych mintermów, należy duplikaty po prostu wyeliminować.

Oto kilka przykładów równań i kodu bajtu LF:

równanie	kod bajtu	równanie	kod bajtu	równanie	kod bajtu
$D = A$	\$f0	$D = A + B$	\$fc	$D = BC$	\$88
$D = a$	\$0f	$D = a + B$	\$cf	$D = Bc$	\$44
$D = B$	\$cc	$D = A + C$	\$fa	$D = bC$	\$22
$D = b$	\$33	$D = a + C$	\$af	$D = bc$	\$11
$D = C$	\$aa	$D = B + C$	\$ee	$D = A+B$	\$f3
$D = c$	\$55	$D = b + C$	\$bb	$D = a+b$	\$3f
$D = AC$	\$a0	$D = AB$	\$c0	$D = A+c$	\$f5
$D = Ac$	\$50	$D = Ab$	\$30	$D = a+c$	\$5f
$D = aC$	\$0a	$D = aB$	\$0c	$D = B+c$	\$dd
$D = ac$	\$05	$D = ab$	\$03	$D = b+c$	\$77

6.3 Określanie obszaru operacji.

Blitter operuje na słowach, a nie na bajtach czy bitach. Jest to bardzo istotne przy określaniu wielkości obszaru operacji blittera.

Jak wcześniej wspomniałem, blitter pracuje na prostokątnych obszarach pamięci.

Na określenie wielkości obszaru operacji składają się:

- określenie modulo (dla blittera, a nie playfieldu),
- określenie wysokości i szerokości operacji.

Przypuśćmy, że chcielibyśmy wyczyścić fragment ekranu o szerokości 320 pikseli i 256 pikseli wysokości, w 2 kolorach. Okno, które chcemy wyczyścić, ma wielkość 96 na 121 punktów.

Najpierw obliczamy modulo (podajemy je w bajtach). Blitter po pobraniu każdego słowa, zwiększa adres zapisany w znacznikach kanałów o 2 bajty. Gdy dojdzie do końca rzędu, wartość modulo jest dodawana do znaczników. Jest to analogiczne do modulo playfieldu.

Szerokość blitu (czyli pojedynczej operacji blittera) w naszym przypadku wynosi 12 bajtów, więc odejmujemy ją od szerokości ekranu w bajtach. Modulo wynosić będzie 28. Wpisujemy je więc do rejestru BLTDMOD (modulo kanału D). Modulo może być także ujemne. A oto adresy rejestrów odpowiadających za modulo wszystkich kanałów.

- BLTCMOD — \$dff060
- BLTBMOD — \$dff062
- BLTAMOD — \$dff064
- BLTDMOD — \$dff066

Do rejestrów tych zapisujemy słowo. Blitter ignoruje najmłodsze bity, ponieważ operuje on na słowach (a nieparzyste modulo doprowadziłoby do nieparzystego adresu).

Teraz określamy wysokość blitu. Mamy 121 linii. Więc czas na określenie szerokości. 96 (czyli szerokość w pikselach) dzielimy przez 16 i mamy wynik: blit ma 6 *słów* szerokości. Teraz wszystko uporządkujemy:

$$WYS \cdot 64 + SZR$$

gdzie:

- WYS to wysokość blitu w liniach,
- SZR to szerokość blitu w *słowach*.

Tak uzyskaną wartość możemy wpisać jako słowo do rejestru BLTSIZE (\$dff058).

Uwaga: należy pamiętać, aby ten rejestr zapisywać jako ostatni (po zdefiniowaniu wszystkich danych dla operacji). Jego zapisanie spowoduje uruchomienie operacji blittera.

Szerokość operacji może przyjąć wartość od 1 do 64 słów, czyli od 16 do 1024 pikseli. Natomiast wysokość operacji może przyjąć wartość od 1 do 1024 linii.

Tryb DESCENDING różni się od normalnego trybu kopiowania tym, że z każdym pobranym słowem, znaczniki kanałów zmniejszają się o dwa bajty, a nie jak dotąd, zwiększały się. Po prostu w trybie tym blitter działa od tyłu. Przesunięcia wykonywane są w lewo, pierwsze słowo maski oddziałowuje na ostatnie słowo rzędu (które wówczas jest pierwszym pobieranym słowem danego rzędu). W tym trybie powinniśmy podawać adres końca obszaru na którym chcemy operować. Wartości modulo, rozmiar blitu kopiowania i innych parametrów nie ulegają zmianie.

6.6 Tryb wypełniania.

Blitter posiada specjalny tryb wypełniania. Umożliwia on na szybsze wypełnienie zawartości ekranu, czy płaszczyzny.

Na jakiej zasadzie blitter wypełnia obszary? Oto najprostszy przykład:

linia przed wypełnieniem:

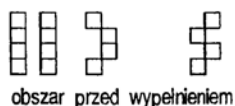
```
%000100000000000000100000000001000000000000000000000000000000100000
```

linia po wypełnieniu:

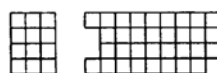
```
%111100000000000011111111110000000000111111111111111100000
```

wypełnianie przebiegało od strony prawej do lewej.

W najprostszym przypadku blitter pobiera kolejne słowa danych do czasu, aż napotka jakiś ustawiony bit. Wtedy pobiera słowa dalej, ale wszystkie zera zamienia na 1. Dzieje się tak, aż do napotkania kolejnego ustawionego bitu. W rezultacie wypełniane są obszary pomiędzy dwoma kolejnymi włączonymi punktami. I tak w kółko, aż zostanie wypełniona pierwsza linia. W związku z tym mechanizmem, blitter posiada specjalny tryb rysowania linii, rysujący jeden piksel w jednej linii ekranu. Ułatwia to znacznie szybkie rysowanie przestrzennej grafiki wypełnianej.



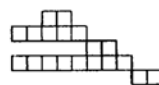
obszar przed wypełnieniem



obszar po wypełnieniu



normalnie rysowana linia



wypełniony odszar ograniczony takimi liniami



linia - jeden punkt w jednym rzędzie



wypełniony odszar ograniczony takimi liniami

W celu wypełnienia używamy standardowego blitu kopiowania, z włączonym trybem descending i ustawiamy bit funkcji EXCLUSIVE lub INCLUSIVE FILL (bity 3 i 4 w rejestrze BLTCON1). Gdy włączymy bit 3, wypełniany będzie obszar pomiędzy liniami, i pozostaną one nietknięte, na przykład %0010001000010000 → %1110001111110000. Włączony bit 4 spowoduje wypełnianie pomiędzy liniami i schowanie linii z lewej strony: %1010001000010000 → %0110000111110000. Bit FILL CARRY IN (także w BLTCON1) spowoduje wypełnianie „na zewnątrz”.

Poszczególne tryby wypełniania można łączyć zgodnie z poniższą tabelką:

EF	IF	FCI
0	1	0
1	0	0
0	1	1
1	0	1

6.7 Używanie wyłączonych kanałów źródłowych

Jeśli nawet wyłączymy kanały A, B czy C, to możemy ich używać w operacjach. Wyłączenie ich powoduje po prostu blokadę dostępu do danych w pamięci. Istnieją 3 rejestry (BLTADAT, BLTBDAT i BLTCDAT) z których blitter pobiera dane (standardowo są one ładowane przez DMA). Na przykład w wypadku wyłączenia kanału A blitter pobiera cały czas tą samą daną zawartą w BLTADAT.

6.8 Bit wykonania i znacznik przerwania

Po zapisaniu rejestru BLTSIZE, blitter ponownie może być wykorzystany tylko wtedy, gdy zakończy poprzednią operację. Służy do tego bit 14 w rejestrze \$dff002. Gdy jest wyzerowany, można z blittera korzystać bez obaw. Jednak gdy jest ustawiony, nie wolno korzystać z blittera, ponieważ jest on jeszcze zajęty wykonywaniem poprzedniej operacji. Oto przykładowa procedura „czekająca” aż blitter skończy pracę.

```
Wait_Blit:    BTST    #14,$dff002
              BNE     Wait_Blit
              RTS
```

Oprócz tego, blitter posiada tzw. znacznik przerwania, który jest ustawiony na 1, gdy blitter zakończy pracę. Więcej informacji o tym znaczniku znajdziesz w rozdziale o przerwaniach.

6.9 Znacznik zerowy

Znacznik zerowy blittera określa, czy operacja logiczna spowodowała ustawienie wszystkich bitów przeznaczenia na 0. Jest to przydatne przy wykrywaniu kolizji obiektów. Bit ten znajduje się w rejestrze DMACONR (bit 13) i jego wartość może być poprawnie interpretowana jedynie w wypadku zakończenia blitu.

6.10 Rysowanie linii

Blitter umożliwia sprzętowe rysowanie linii. Aby uaktywnić tryb liniowy, trzeba ustawić bit 0 w BLTCON1.

Uwaga: ustawienie tego bitu powoduje zmianę znaczenia niektórych bitów w rejestrach BLTCON0 i BLTCON1.

W trybie liniowym blitter rysuje linię od jednego do drugiego zadanego punktu. Nie wystarczy jednak podanie „surowych” współrzędnych. Trzeba je niestety odpowiednio przerobić i wynikowe wartości wpisać do odpowiednich rejestrów. Najpierw jednak poznamy znaczenie rejestrów BLTCON0 i BLTCON1 w trybie liniowym:

BLTCON0:

bity 15 – 12 — START3 – 0 — cztery najmłodsze bity pozycji startowej X,
 bit 11 — USEA — czy używany jest port A,
 bit 10 — USEB — czy używany jest port B,
 bit 9 — USEC — czy używany jest port C,
 bit 8 — USED — czy używany jest port D,
 bity 7 – 0 — LF7 – LFO — określają rodzaj operacji.

BLTCON1:

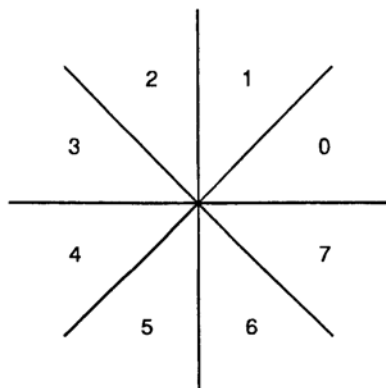
bity 15 – 12 — TXT3 – TXT0 — przesunięcie portu B,
 bity 11 – 7 — — nie używane,
 bit 6 — SIGN — ustawiany, gdy $2 * Dy < Dx$,
 bit 5 — — nie używany,
 bit 4 — SUL
 bit 3 — SUD — te trzy bity (tj. 4, 3, 2) określają oktant,
 bit 2 — AUL
 bit 1 — SING — linie do wypełnienia,
 bit 0 — LINE — ustawiony na 1 (odpowiada za tryb kreślenie linii).

Przypuśćmy, że chcemy wykreślić linię od punktu X1, Y1 do X2, Y2. Musimy więc najpierw wyznaczyć oktant, czyli kierunek kreślenia linii, a także różnicę pomiędzy X1 a X2 i Y1 a Y2 (musi być ona dodatnia). Od teraz będziemy je nazywać odpowiednio DeltaX i DeltaY.

$$\text{DeltaX} = |X1 - X2|$$

$$\text{DeltaY} = |Y1 - Y2|$$

Kiedy znamy już te wartości, możemy obliczyć oktant. Oto schematyczny rysunek rozmieszczenia oktantów:



Bity 4 – 2 w rejestrze BLTCON1 powinny przyjmować następujące wartości w zależności od oktantu:

bity			numer oktantu
4	3	2	
1	1	0	0
0	0	1	1
0	1	1	2
1	1	1	3
1	0	1	4
0	1	0	5
0	0	0	6
1	0	0	7

Po wyznaczeniu oktantu musimy jeszcze określić dwie wartości D_y i D_x . D_y to mniejsza wartość z ΔX i ΔY , a D_x — większa. Jeśli teraz $2 * D_y < D_x$, to ustawiany bit SIGN w BLTCON1. Ustawiamy teraz rejestr adresowy kanału A na wartość $(4 * D_y) - (2 * D_x)$ oraz modulo kanału A na $4 * (D_y - D_x)$, a modulo kanału B na $4 * D_y$.

Do rejestru danych A, który jest w tym wypadku używany jako rejestr indeksowy, wpisujemy wartość \$8000 (BLTADAT). Oba słowa maski ustawiamy \$ffff. Do bitów START 3 – 0 wpisujemy X1 and 15.

Rejestr danych B powinien zawierać wzór linii (\$ffff — dla linii ciągłej). Przesunięcie portu B (bity TXT3 – TXT0 w BPLCON1) powinno przyjąć numer bitu w słowie od którego rozpoczynamy rysowanie linii.

Rejestry adresowe źródeł C i D powinny zawierać adres słowa zawierającego pierwszy piksel linii, natomiast modulo rejestrów C i D powinny mieć wartość szerokości ekranu, na którym rysujemy linię.

Bity USEA, USEC i USED powinny być ustawione na 1, a USEB na 0. Jeśli chcemy kreślić linie do wypełniania, to musimy także ustawić bit SING (nie mylić SIGN!), w BLTCON1.

Kanał A reprezentuje punkt do ustawienia w linii, C to źródło ekranu, B — wzór linii do narysowania. Więc operacja stawiania linii, będzie miała kombinację

$$AB + aC$$

Wysokość blitu ustalamy na $Dx + 1$, a szerokość na dwa słowa.

Przykładowa procedura rysowania linii (do wypełniania) zamieszczona jest w rozdziale 10. Standardowo bajt LF jest ustawiony na \$4a (funkcja EOR). Aby procedura rysowała zwykłe linie, należy zmienić \$4a na \$ca, a także wyłączyć tryb rysowania linii dla wypełniania.

6.10.1 Podsumowanie wiadomości o rysowaniu linii

Oto kroki, które należy wykonać, aby narysować linię o współrzędnych $x1, y1$ i $x2, y2$.

Wykonujemy początkowe obliczenia:

$$\Delta X = |X1 - X2|$$

$$\Delta Y = |Y1 - Y2|$$

$$Dx = \max(\Delta X, \Delta Y)$$

$$Dy = \min(\Delta X, \Delta Y)$$

Potem odpowiednio ustawiamy rejestry:

Rejestr	Ustawienie
BLTADAT	\$8000
BLTBDAT	\$ffff — wzór linii
BLTAFWM	\$ffff — maska
BLTALWM	\$ffff — maska
BLTAMOD	$4 * (Dy - Dx)$
BLTBMOD	$4 * Dy$
BLTCMOD	szerokość ekranu w bajtach
BLTDMOD	szerokość ekranu w bajtach
BLTAPT	$(4 * Dy) - (2 * Dx)$
BLTBPT	nie używany
BLTCPT	adres słowa zawierającego pierwszy piksel linii
BLTDPT	adres słowa zawierającego pierwszy piksel linii

Rejestr	Ustawienie
BLTCON0	bity 15 – 12 X1 AND 15 bity USEA, USEC, USED 1 bit USEB 0

Ustawiamy kombinację na AB + aC

Rejestr	Ustawienie
BLTCON1	bit LINE 1 bity 4 – 2 numer oktantu bity 15 – 12 numer bitu dla początku wzoru linii bit SIGN 1 jeśli BLTAPT jest ujemny 0 w przeciwnym przypadku bit SING 1 jeśli linia ma mieć 1 piksel na linię 0 w przeciwnym przypadku
BLTSIZE	$Dx + 1 * 64 + 2$

6.11 Reguła programowania blittera

Oto lista „przykazań”, których trzeba przestrzegać przy pracy z blitterem:

- zanim użyjesz blittera, sprawdź czy przypadkiem nie wykonuje jeszcze jakiegś operacji,
- modulo i adresy podajemy w bajtach, szerokość w słowach, a wysokość w pikselach,
- rejestr BLTSIZE powinien być zapisywany jako ostatni,
- wypełnianie obszarów powinno się odbywać tylko w trybie descending,
- w normalnym trybie blitter zwiększa adresy, dodaje modulo i przesuwa w prawo,
- w trybie descending blitter zmniejsza adresy, odejmuje modulo i przesuwa w lewo.

6.12 Kontrolowanie blittera za pomocą coppera.

Przy programowaniu grafiki przestrzennej, bardzo dużo czasu zabiera oczekiwanie, aż blitter zakończy pracę, w celu jego ponownego wykorzystania. Metoda ta powoduje, że procesor główny beczynnie oczekuje w pętli na zakończenie blitu. Istnieje jednak metoda umożliwiająca kontrolowanie blittera copperem, która od mikroprocesora wymaga jedynie ułożenia odpowiedniej copperlisty. Copper będzie wtedy zapisywać rejestry blittera i startować go, a także on będzie oczekiwał na zakończenie blitu. Motorola ograniczać się będzie tylko do kontrolowania copperlisty.

Aby zaprząć coppera do kontroli nad blitterem, musimy najpierw zezwolić mu na korzystanie z rejestrów tego drugiego. Dokonujemy tego ustawiając na 1 bit CDANG w rejestrze COPCON. Dopiero wtedy układamy odpowiednią copperlistę. Rozkazem WAIT coppera z włączonym bitem blittera oczekujemy na zakończenie blitu. Rozkaz ten może wyglądać tak:

```
DC.L    $00010000
```

Dopiero po wykonaniu tej instrukcji mamy prawo ponownego użycia blittera. A oto fragment copperlisty kasujący i wypełniający blitterem ekran 320×256 w jednym bitplanie pod adresem \$60000.

```
...
...
...
DC.L    $00010000
DC.L    $00540006
DC.L    $00560000
DC.L    $00660000
DC.L    $00400100
DC.L    $00420000
DC.W    $0058,256*64+20
...
...
...
DC.L    $00010000
DC.L    $004009f0
DC.W    $0042,%01010
DC.L    $0044ffff
DC.L    $0046ffff
DC.L    $00500006
DC.L    $005227ff
DC.L    $00540006
DC.L    $005627ff
DC.L    $00640000
DC.L    $00660000
DC.W    $0058,256*64+20
...
...
...
```

6.13 Przykłady programowania blittera.

Jednym z zadań jakie blitter może bardzo szybko wykonywać, to czyszczenie pamięci. W tym celu możemy użyć tylko jednego kanału — docelowego D. Oto przykładowa procedura czyszcząca ekran o rozmiarach 320×256 punktów w jednym bitplanie:

```
-----
Clear_Screen:  LEA      $dff000,a6      ;baza rejestrów sprzętowych w A6

               Wait_Blit: BTST      #14,$02(a6)      ;czy blitter zakończył poprzedni blit?
                  BNE      Wait_Blit      ;jeśli nie to czekaj dalej...

                  MOVE.L    #Screen,$54(a6)          ;adres kanału D
                  MOVE.W    #$0000,$66(a6)          ;modulo D=0
                  MOVE.W    #%00000000100000000,$40(a6)
;używamy tylko kanału D i bez żadnej kombinacji mintermów (BLTCON0)
                  MOVE.W    #$0000,$42(a6)          ;BLTCON1=0
                  MOVE.W    #256*64+20,$58(a6)      ;start blitu (256 linii
                                                    ;na 20 słów (320 punktów))

                  RTS

Screen:        BLK.B    40*256
-----
```

Następnym przykładem będzie zwykłe kopiowanie pamięci. W tym celu musimy użyć funkcji $A = D$. Kopiuje ona obszar 320×256 punktów w jednym bitplanie spod etykiety **Screen1** do **Screen2**.

```

;-----
Copy_Block:
    LEA    $dff000,a6
    BSR    Blit_wait
    MOVE.W #0,$64(a6)
    MOVE.W #0,$66(a6)
    MOVE.L #$09f00000,$40(a6)
    MOVE.L #$ffffff,$44(a6)
    MOVE.L #Screen1,$50(a6)
    MOVE.L #Screen2,$54(a6)
    MOVE.W #256*64+20,$58(a6)
    RTS

Wait_Blit:
    BTST   #14,$02(a6)
    BNE    Wait_Blit
    RTS

Screen1: BLK.B $2800
Screen2: BLK.B $2800

```

Jeśli teraz będziemy chcieli także wypełnić nasz obszar, to włączamy tryb DESCENDING i tryb wypełniania. Zamiast **Screen1** podajemy **Screen1+\$2800-1**, czyli koniec obszaru. Oto przykład:

```

Fill_Screen:
    LEA    $dff000,a6
    BSR    Blit_wait
    MOVE.W #0,$64(a6)
    MOVE.W #0,$66(a6)
    MOVE.W #$09f0,$40(a6)
    MOVE.W #%01010,$42(a6)
    MOVE.L #$ffffff,$44(a6)
    MOVE.L #Screen1+$2800-1,$50(a6)
    MOVE.L #Screen1+$2800-1,$54(a6)
    MOVE.W #256*64+20,$58(a6)
    RTS

Wait_Blit:
    BTST   #14,$02(a6)
    BNE    Wait_Blit
    RTS

Screen1: BLK.B $2800

```

Teraz możemy zająć się przesuwaniem zawartości ekranu zarówno w lewo, jak i w prawo. Wykorzystujemy w tym celu standardowy blit kopiowania i ustawiamy w nim przesunięcie kanału A. Scrollowanie całego ekranu w prawo będzie wyglądać następująco:

```

Copy_Block:
    LEA    $dff000,a6
    BSR    Blit_wait
    MOVE.W #0,$64(a6)
    MOVE.W #0,$66(a6)
    MOVE.L #$19f00000,$40(a6)      ;przesuwamy o jeden punkt
    MOVE.L #$ffffff,$44(a6)
    MOVE.L #Screen1,$50(a6)
    MOVE.L #Screen2,$54(a6)
    MOVE.W #256*64+20,$58(a6)
    RTS

Wait_Blit:
    BTST   #14,$02(a6)
    BNE    Wait_Blit
    RTS

Screen1: BLK.B $2800

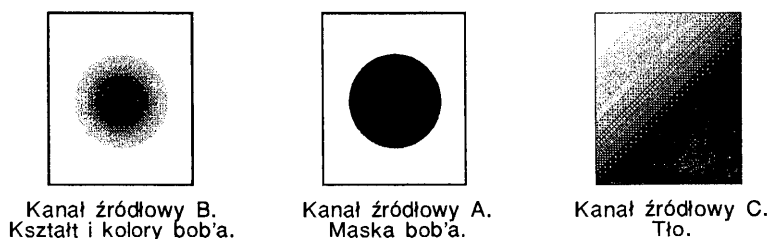
```

Przy pisaniu procedury scrolla poziomego (płynących napisów), musimy scrollować od prawej do lewej. W tym celu musimy włączyć tryb DESCENDING i zamiast początku, podawać koniec obszaru do kopiowania.

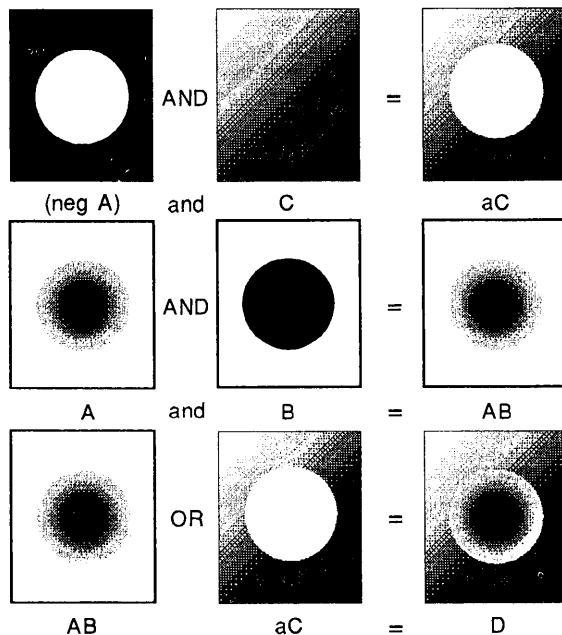
Kolejnym problemem jakim się zajmujemy, to *boby* (ang. *blitter objects*). Są to obiekty graficzne „narzucane” na ekran.

Aby narzucić boba na ekran, musimy usunąć wszystko to, co znajduje się w tym miejscu, w którym chcemy obiekt postawić. W naszym wypadku możemy użyć jednej operacji: $D = AB + aC$, gdzie A to tzw. maska obiektu, B — rysunek obiektu (boba), a C — tło (ekran). Czym jest maska? Jest to po prostu obiekt na wszystkich bitplanach. Jeśli chcemy stworzyć maskę dla statku kosmicznego w dwóch bitplanach, to rysujemy ten sam statek w jednym kolorze, który daje kombinację %11 na tych bitplanach. Będzie to kolor nr 3. W przypadku pięciu bitplanów będzie to kolor nr 31.

Teraz zastanówmy się co się będzie działo z tłem i obiektem przy kombinacji $D = AB + aC$. A and B daje w rezultacie obiekt, a a and C daje całe tło otaczające obiekt. W wyniku funkcji or, otrzymujemy tło z narzuconym obiektem.



Interpretacja graficzna równania $D = AB + aC$



Narzucanie obiektu prezentuje poniższy przykład. Stawia on na ekranie boba o współrzędnych (X; Y), o wymiarach 32 na 32 punkty. Ponieważ istnieje możliwość przesunięcia, obrazek i maska boba w pamięci powinny mieć szerokość 48 punktów. Zarówno ekran i bob muszą być w trybie RAWBLIT.

```

;-----
;Stawianie Boba na ekran
; D0 - pozycja X, D1 - pozycja Y, A0 - Bob, A1 - Maska, A2 - Ekran

Bob_Draw:      LEA      $dff000,a6
                MULS    #40*2,D1
                ADDA.L  D1,A2
                MOVEQ   #0,D1
                MOVE.W  D0,D1
                AND.W   #$f,d0
                LSL.W   #8,D0
                LSL.W   #4,D0
                AND.W   #$fff0,d1
                LSR.W   #3,D1
                ADD.L   D1,A2
                MOVE.W  #$fca,d2
                OR.W    D0,D2
                BSR     Wait_Blit
                MOVE.W  D2,$40(a6)
                MOVE.W  D0,$42(a6)
                MOVE.L  A2,$48(a6)
                MOVE.L  A0,$4c(a6)
                MOVE.L  A1,$50(a6)
                MOVE.L  A2,$54(a6)
                MOVE.W  #40-6,$60(a6)
                MOVE.W  #40-6,$66(a6)
                MOVE.W  #0,$62(a6)
                MOVE.W  #0,$64(a6)
                MOVE.W  #32*2*64+3,$58(a6)
                RTS

                ;przesunięcie A
                ;przesunięcie B
                ;pointer C      -      tło
                ;pointer B      -      bob
                ;pointer A      -      maska
                ;pointer D      -      tło
                ;modulo C
                ;modulo D
                ;modulo C
                ;modulo D

Wait_Blit:     BTST    #14,$02(a6)
                BNE     Wait_Blit
                RTS
;-----

```


Rozdział VII

Spritey

7.1 Wprowadzenie

Spritey (zwane „duszkami”) są doskonale znane posiadaczom mniejszych komputerów (np. C64). Czym one dokładnie są? Inaczej mówiąc, są to po prostu „niezależne” obiekty graficzne, które można wyświetlać w dowolnym miejscu na ekranie, przy czym pamięć ekranu pozostaje niezmienną. Są one obsługiwane przez system, co umożliwia ich swobodną animację, itp.

Amiga posiada 8 kanałów DMA, które obsługują spritey. Oczywiście nikt nam nie broni obsługiwać ich „ręcznie” (np. za pomocą coppera, czy mikroprocesora).

Podstawowe spritey mają 16 pikseli szerokości i mogą być wysokie na dowolną liczbę linii. Mogą one być w 3 kolorach, plus 1 przezroczysty.

Każdy z kanałów DMA może być użyty kilkakrotnie w czasie jednego wyświetlania. Oznacza to, że nie jesteśmy ograniczeni posiadaniem 8 spriteów na ekranie. Dzięki odpowiedniemu postępowaniu, można ich wyświetlić na ekranie nawet kilkaset.

7.2 Definiowanie struktury spritea dla kanałów DMA.

Aby wyświetlić spritea za pomocą kanałów DMA, należy stworzyć jego strukturę w pamięci, a potem zapisać jej adres do odpowiednich rejestrów sprzętowych. Należy pamiętać, że każdy kanał DMA ma osobne adresy dla struktury spriteów. Wynika z tego, że jeśli każdemu kanałowi możemy przypisać jedną strukturę, to możemy na raz wyświetlić spritey z ośmiu struktur.

Definiowanie spritea polega na określeniu jego cech:

- wysokości,
- kształtu.
- kombinacji kolorów,
- ewentualnego połączenia spriteów,
- pozycji na ekranie.

7.2.1 Definiowanie pozycji

Pozycję spritea na ekranie jest określona parą współrzędnych X i Y. Pozycja (0; 0) ($X = 0$ i $Y = 0$), znajduje się w lewym górnym rogu ekranu. Współrzędne X i Y w Amigach bez układów AGA zawsze dotyczą niskiej rozdzielczości.

Jednak pozycja (0; 0) nie jest normalnie widoczna. Znajduje się ona poza widoczną częścią playfieldu. Wielkość obszaru, w którym może być wyświetlony sprite, ograniczona jest także rozmiarem okna playfieldu.

Przy określaniu pozycji spritea, *zawsze* podajemy współrzędne punktu znajdującego się w jego lewym, górnym rogu.

7.2.1.1 Definiowanie pozycji poziomej

Pozycja pozioma spritea, może zawierać się między 0 a 447 pikselem ekranu. Aby jednak był on widoczny, musi znajdować się w obszarze okna wyświetlania. Mamy na przykład okno o szerokości 320 punktów: od pozycji 64 do 384. Jeśli teraz pozycja będzie znajdować się poza lewą lub prawą pozycją okna wyświetlania, to sprite nie będzie widoczny (lub widoczny częściowo).

Z tego możemy teraz wyprowadzić prosty wzór na obliczenie właściwej pozycji ekranu. Traktujemy lewy, górny róg playfieldu jako punkt (0;0). Rzeczywista pozycja spritea będzie więc wynosić:

$$RX = PZX1 + PZX2$$

gdzie:

- RX — rzeczywista pozycja X spritea,
- PZX1 — pozycja X spritea względem początku okna,
- PZX2 — pozycja X początku ekranu wyświetlania.

7.2.1.2 Definiowanie pozycji pionowej

Dla pozycji pionowej spritea, możemy wybrać dowolną linię od 0 do 312. Postępujemy analogicznie jak do pozycji poziomej:

$$RY = PZY1 + PZY2$$

gdzie:

- RY — rzeczywista pozycja Y spritea,
- PZY1 — pozycja Y spritea względem początku okna,
- PZY2 — pozycja Y początku okna wyświetlania.

7.2.2 Definiowanie rozmiaru spriteów

Spritey mają szerokość 16 pikseli i nie można tego zmienić. Natomiast ich wysokość może być praktycznie dowolna. Może wynosić np. 1 lub nawet wysokość całego ekranu. Spritey są niezależne od trybu wyświetlania playfieldu, tak więc zmiany np. rozdzielczości ekranu, nie wpływa na wielkość spritea.

7.2.3 Kształt spriteów

Możemy zdefiniować dowolny kształt spritea. Wiele TIFF—CONVERTERów, umożliwia nagranie fragmentu danego obrazka w postaci spritea. Oto jednak prosty przykład ręcznego definiowania kształtu spritea.

Rysujemy, na kartce kontur spritea i określamy, które punkty mają być włączone, a które wyłączone. Kształt „dopełniamy” zerami, czyli kolorem przezroczystym do szerokości 16 punktów.

1	00000010000000
111	00000111000000
111	00000111000000
1111	00000111100000
11111	00001111100000
11111	00001111100000
111111	00001111110000
1111111	00011111110000
11111111	00011111111000
1111111111	00111111111100
11	00000011000000
11	00000011000000
1111	00000111100000

7.2.4 Kolory spriteów

Gdy mamy zdefiniowany kształt spritea, to musimy określić jego kolory. W normalnym trybie wyświetlania, sprite może być trójkolorowy. Do zdefiniowania koloru, potrzeba dwóch słów na jedną linię (szerokość wynosi 16 bitów, a 2^2 wynosi 4).

pierwsze słowo	%1000000000000001
drugie słowo	%0000000100000001
numery kolorów	#1000000200000003

Otrzymana linia daje nam w rezultacie numery rejestrów kolorów, z których ma skorzystać Amiga przy wyświetlaniu spritea.

Kombinacja %00 ma tutaj szczególne znaczenie. W miejscu piksela o tej kombinacji bitów, będzie wyświetlane to, co znajduje się pod spodem. Tak więc, jeśli sprite będzie miał większy priorytet wyświetlania od tła (będzie „nad” playfieldem), to w tym miejscu będzie wyświetlane tło.

Amiga w przeciwieństwie do innych komputerów (np. C64), nie posiada osobnych rejestrów barw dla spriteów. Korzysta one z tych samych rejestrów barw zarówno dla playfieldu, jak dla spriteów. Może to wywoływać komplikacje z doбором kolorów, gdy będziemy chcieli wyświetlić playfield w 32 lub 64 kolorach.

Spritey są pogrupowane po dwa i każda para ma przyporządkowane 4 rejestry barw.

Sprite	Rejestr	Kolor
Spritey 0 i 1	COLOR16	przezroczysty
	COLOR17	kombinacja %01
	COLOR18	kombinacja %10
	COLOR19	kombinacja %11
Spritey 2 i 3	COLOR20	przezroczysty
	COLOR21	kombinacja %01
	COLOR22	kombinacja %10
	COLOR23	kombinacja %11
Spritey 4 i 5	COLOR24	przezroczysty
	COLOR25	kombinacja %01
	COLOR26	kombinacja %10
	COLOR27	kombinacja %11
Spritey 6 i 7	COLOR28	przezroczysty
	COLOR29	kombinacja %01
	COLOR30	kombinacja %10
	COLOR31	kombinacja %11

7.2.5 Tworzenie struktury spritea

Aby poinformować dany kanał DMA jak ma być wyświetlany sprite, musimy utworzyć w pamięci jego strukturę. Aby ją stworzyć, należy:

- zapisać poziomą i pionową pozycję spritea,
- zapisać pionową pozycję końca spritea,

- odpowiednio zakodować i zapisać kolory oraz kształt spritea,
- odpowiednio oznaczyć koniec struktury spritea.

7.2.5.1 Pierwsze słowo struktury

Słowo to zawiera pionową i poziomą (VSTART i HSTART) pozycję spritea na ekranie. Oto znaczenie kolejnych bitów.

bity 15 – 8 — bity 7 – 0 pozycji VSTART,

bity 7 – 0 — bity 1 – 8 pozycji HSTART.

7.2.5.2 Drugie słowo struktury

Słowo to zawiera pionową pozycję końcową (VSTOP) spritea, bity 8 i 0 pozycji VSTART i HSTART. Oprócz tego zapisujemy w nim bit, który umożliwia uzyskanie 16 kolorowych spriteów. Zostanie to jednak omówione nieco później. Oto znaczenie kolejnych bitów.

bity 15 – 8 — bity 7–0 pozycji VSTOP,

bit 7 — tryb 16 kolorowych spriteów (ATTACH),

bity 6 – 3 — nie używane (skasować),

bit 2 — bit 8 pozycji VSTART,

bit 1 — bit 8 pozycji VSTOP,

bit 0 — bit 0 pozycji HSTART.

Jak można zauważyć, wartość VSTOP – VSTART określa wysokość spritea.

7.2.5.3 Zapis koloru i kształtu spritea

Jak wcześniej wspomniałem, do zakodowania kolorów spritea, używamy dwóch słów. Rozbijamy więc numery kolorów na liczby binarne:

0102030302010203

%0100010100010001 — starsze słowo

%0001010101000101 — młodsze słowo

Analogicznie postępujemy ze wszystkimi liniami spritea i zapisujemy w następującej kolejności:

```
starsze słowo linii 1
młodsze słowo linii 1
starsze słowo linii 2
młodsze słowo linii 2
starsze słowo linii 3
młodsze słowo linii 3
```

```
starsze słowo linii n
młodsze słowo linii n
```

Pamiętajmy, by zapisać w strukturze tyle linii ile wynosi wartość VSTOP–VSTART.

7.2.5.4 Ponowne użycie spritea

Aby dany kanał DMA mógł wyświetlić kilka spriteów jednocześnie, należy w strukturze po ostatniej linii spritea, napisać kolejną strukturę dla następnego.

Metoda ta jednak ma pewne ograniczenie. Każdy kanał DMA spriteów odczytuje dwa słowa na jedną linię rastra i wyświetla je na bieżąco, wraz z wiązką promienia elektronowego. Oznacza to, że ponownie użycie tego samego kanału w celu wyświetlenia dodatkowych spriteów będzie możliwe tylko wtedy, gdy początkowa pozycja pionowa kolejnego spritea będzie większa o *co najmniej* jedną linię od końcowej pozycji pionowej poprzedniego.

7.2.5.5 Znacznik końca struktury spritea

Żeby DMA mógł zakończyć pobieranie danych po pobraniu wszystkich linii z danej struktury, DMA sprawdza czy następne długie słowo jest równe \$00000000. Jeśli tak, DMA kończy pobieranie danych. W przeciwnym wypadku traktuje je jako pierwsze i drugie słowo kontroli następnego spritea i stara się wyświetlić go na ekranie.

7.3 Wyświetlanie spriteów

Gdy struktura spritea zostanie już zdefiniowana, to aby go wyświetlić, należy podać systemowi niezbędne parametry. Należą do nich:

- decyzja na jeden z ośmiu kanałów DMA spriteów (możemy użyć też kilka na raz),
- ewentualne uruchomienie danego kanału,
- przekazanie adresu początku struktury do odpowiedniego rejestru,
- co każdy okres wygaszania pionowego, zapisywać od nowa adresy struktur.

Należy także pamiętać o tym, że aby wyświetlić spritea, musi być włączony *co najmniej* jeden bitplan. W przeciwnym wypadku nic na ekranie nie zobaczymy.

7.3.1 Wybieranie kanału DMA i określanie adresów

Każdy kanał DMA spriteów posiada własną parę rejestrów adresowych, do których wpisujemy adres początku struktury spritea lub spriteów. Oto ich adresy:

- SPR0PTH — \$dff120 — 3 górne bity adresu struktury dla DMA nr 0,
- SPR0PTL — \$dff122 — 16 dolnych bity adresu struktury dla DMA nr 0,
- SPR1PTH — \$dff124 — 3 górne bity adresu struktury dla DMA nr 1,
- SPR1PTL — \$dff126 — 16 dolnych bity adresu struktury dla DMA nr 1,
- SPR2PTH — \$dff128 — 3 górne bity adresu struktury dla DMA nr 2,

- SPR2PTL — \$dff12a — 16 dolnych bity adresu struktury dla DMA nr 2,
- SPR3PTH — \$dff12c — 3 górne bity adresu struktury dla DMA nr 3,
- SPR3PTL — \$dff12e — 16 dolnych bity adresu struktury dla DMA nr 3,
- SPR4PTH — \$dff130 — 3 górne bity adresu struktury dla DMA nr 4,
- SPR4PTL — \$dff132 — 16 dolnych bity adresu struktury dla DMA nr 4,
- SPR5PTH — \$dff134 — 3 górne bity adresu struktury dla DMA nr 5,
- SPR5PTL — \$dff136 — 16 dolnych bity adresu struktury dla DMA nr 5,
- SPR6PTH — \$dff138 — 3 górne bity adresu struktury dla DMA nr 6,
- SPR6PTL — \$dff13a — 16 dolnych bity adresu struktury dla DMA nr 6,
- SPR7PTH — \$dff13c — 3 górne bity adresu struktury dla DMA nr 7,
- SPR7PTL — \$dff13e — 16 dolnych bity adresu struktury dla DMA nr 7.

Należy zawsze pamiętać, by po włączeniu DMA spriteów, do nieużywanych kanałów zapisywać tzw. „fikcyjne” struktury, czyli spritey bez zdefiniowanego kształtu, o wysokości równej 0 linii.

W naszym wypadku taka struktura wystarczy:

```
Fikcyjna_Struktura:      DC.L      0,0
```

Jednak po co to robić? Otóż należy pamiętać, że w rejestrach adresowych struktur, nieużywanych przez nas, mogą znajdować się najróżniejsze wartości. Wskazywać one mogą wszystko, więc na ekranie mogą pojawiać się i znikać różne fragmenty pamięci, interpretowanych jako spritey.

7.3.2 Odświeżanie adresu struktury

Zawartość rejestrów SPRxPT jest zwiększana po każdym pobraniu długiego słowa, o 4 bajty. Tak samo dzieje się z pointerami (zawartością rejestrów adresowych) bitplanów, itp. Dzieje się tak, ponieważ DMA musi wiedzieć, spod jakiego adresu ma pobrać następną daną.

W tym wypadku, co każdy okres wygaszania pionowego, należy odświeżać (zapisywać od nowa) adresy struktur. Najwygodniejszą metodą jest ułożenie odpowiedniej copperlisty. Dzięki niej, będziemy mieli pewność, że adresy struktur spriteów, będą odświeżane co każdą ramkę.

7.4 Priorytety spriteów

Każdy sprite ma określony priorytet wyświetlania, względem każdego innego. Zasada jest bardzo prosta. Im wyższy numer spritea, tym niższy ma on priorytet.

7.5 Łączenie spriteów

Aby uzyskać szerszego spritea, należy połączyć dwa. Jeden musi być wtedy oddalony od drugiego w poziomie o 16 punktów. Należy pamiętać, by podczas ich animacji to uwzględnić.

7.5.1 Spritey 16 kolorowe

Spritey można także łączyć w celu uzyskania większej ilości kolorów. Normalnie mamy dostępne 3 kolory i przezroczysty. Jednak nakładając dokładnie dwa spritey na siebie, mamy dostępne 4 słowa na jedną linię czyli $2^4 = 16$ kombinacji, ponieważ jednemu punktowi przypisane są 4 bity. Daje to w sumie 15 kolorów i przezroczysty.

Aby uaktywnić ten tryb, musimy stworzyć dwie struktury dla oddzielnych kanałów DMA. Należy także ustawić w jednej z nich (tej o numerze nieparzystym) bit ATTACH (p. rozdz. 7.2.5.2). Możliwe są następujące połączenia:

- sprite 1 ze spritem 0,
- sprite 3 ze spritem 2,
- sprite 5 ze spritem 4,
- sprite 7 ze spritem 6.

Jak w tym wypadku tworzony jest obraz spritea? Jeśli połączymy na przykład spritea 1 ze spritem 0, to kolor jest definiowany tak:

Starsze słowo spritea 1	% 0 0 1 0 0 0 1 1 0 1 1 0 1 0 0 0
Młodsze słowo spritea 1	% 1 0 1 0 1 0 0 0 1 1 1 1 0 0 0 1
Starsze słowo spritea 0	% 0 1 0 0 1 0 0 0 1 1 1 0 1 1 0 0
Młodsze słowo spritea 0	% 0 0 1 0 0 1 0 1 1 0 1 1 0 0 1 0
Wynik	2 4 11 0 3 8 1 9 14 7 15 10 5 4 8 2

Jak widać, w kombinacji najmniej znaczącym jest bit pobrany ze starszego słowa spritea o numerze nieparzystym.

Po dodaniu do numeru koloru każdego punktu z linii liczby 16, wiemy z jakiego rejestru kolorów Amiga ma pobrać kolor w czasie wyświetlania danego punktu. Oto tabelka kombinacji kolorów i wykorzystywanych rejestrów.

dziesiętna wartość kombinacji	wartość binarna	numer rejestru koloru
0	%0000	16
1	%0001	17
2	%0010	18
3	%0011	19
4	%0100	20
5	%0101	21
6	%0110	22

dziesiątna wartość kombinacji	wartość binarna	numer rejestru koloru
7	%0111	23
8	%1000	24
9	%1001	25
10	%1010	26
11	%1011	27
12	%1100	28
13	%1101	29
14	%1110	30
15	%1111	31

Przy tworzeniu spriteów 16 kolorowych należy pamiętać, by dokładnie się pokrywały. Należy także uważać w czasie animacji, ponieważ przesunięcie o jeden piksel jednego z nich, spowodować może powstanie barwnej mozaiki.

7.6 Tryb ręczny obsługi spriteów

Zwykle spritey są obsługiwane automatycznie przez DMA, ale nic nie stoi na przeszkodzie, aby je wykorzystać bez użycia kanałów. Najlepiej do tego celu wykorzystać coppera, pisząc odpowiednią copperlistę. Aby spritea wyświetlić, musimy skorzystać z następujących rejestrów:

- SPRxPOS — \$dff140 — Pozycja pion.—poz. spritea x; start,
- SPRxCTL — \$dff142 — Pozycja pionowa końca spritea x,
- SPRxDATA — \$dff144 — Rejestr A danej obrazu spritea x,
- SPRxDATB — \$dff146 — Rejestr B danej obrazu spritea x.

Ponieważ w trybie ręcznym pozycja pionowa nie ma znaczenia, musimy w odpowiedniej pozycji promienia zapisać wyżej wymienione rejestry. SPRxDAT odpowiadają zakodowanej linii obrazu spritea w strukturze dla DMA. Musimy także ustalić jeszcze pozycję poziomą, i sprite będzie wyświetlany. Jednak powstanie wtedy tylko długi, kolorowy, pionowy pasek. Aby wyświetlić spritea np. o wysokości 16 linii, to zadanie copperlisty będzie polegać na zmianie co linię zawartości rejestrów SPRxDAT. Jednak nie byłoby zupełnie sensu, ponieważ to samo możemy zrobić w normalnym trybie! Ale tryb ręczny oferuje nam możliwość zmiany pozycji poziomej każdej linii spritea także wielokrotne użycie tego samego spritea w jednej linii. Jak to działa, można się przekonać oglądając efekt TWIST—SCROLLa w demie HARDWIRED.

Uwaga: zanim zastosujemy tryb ręczny obsługi spriteów, musimy najpierw wyłączyć DMA spriteów!

7.7 Kolizje między spriteami, a playfieldem

Możliwe jest wykrywanie kolizji między grupami spriteów, a także pomiędzy spriteami i playfieldami. Jednak dokładny opis znajduje się w rozdziale poświęconym kontroli systemu.

7.8 Podsumowanie wiadomości o rejestrach obsługi spriteów

Istnieje 8 grup rejestrów do obsługi spriteów. W skład każdej z nich wchodzi 6 rejestrów.

- Rejestry adresowe (SPRxPTH i SPRxPTL) — używane są one w celu wskazania systemowi początku struktury spritea dla danego kanału DMA. Ich wartość po pobraniu danej przez kanał jest zwiększana tak, by wskazywała następną daną w pamięci. Tak więc wymagane jest przywracanie poprzedniej zawartości tych rejestrów.
- Rejestry kontrolne (SPRxPOS i SPRxCTL) — określają pozycję i tryb wyświetlania danego spritea. Pierwszy z nich (SPRxPOS) zawiera dane pozycji lewego, górnego piksela spritea. Oto znaczenie poszczególnych bitów tego rejestru:

bity 15 – 8 — bity 7 – 0 pozycji VSTART,

bity 7 – 0 — bity 1 – 8 pozycji HSTART.

SPRxCTL zawiera informacje kontrolne dla procesu pobierania danych. Oto jego znaczenie:

bity 15 – 8 — bity 7 – 0 pozycji VSTOP,

bit 7 — tryb 16 kolorowych spriteów (ATTACH),

bity 6 – 3 — nie używane (skasować),

bit 2 — bit 8 pozycji VSTART,

bit 1 — bit 8 pozycji VSTOP,

bit 0 — bit 0 pozycji HSTART.

- Rejestry danych (SPRxDATA i SPRxDATB) — zapisywane są tutaj dane dla pojedynczej linii spritea. Zwykle ładowane są one automatycznie przez kanały DMA (zapisywane są tutaj kolejne linie struktury), ale można tego dokonywać za pomocą coppera, czy Motoroli.

7.9 Przykład procedury obsługującej spritea

Poniżej przedstawiam procedurę spritea 16x16 punktów, „latającego” po ekranie i odbijającego się od jego boków. Pozycja spritea na ekranie jest zawarta pod etykietami **Sprx** i **Spry**. Dopiero wywołanie procedury **Sprite_Position** powoduje konwersję i zapisanie pozycji do struktury spritea. Myślę, że analiza tej procedury nie powinna przysporzyć problemów.

```

X_Offs_Spr      =      128
Y_Offs_Spr      =      44
Wys_Spr         =      16
Xmin_Spr        =      0
Ymin_Spr        =      0
Xmax_Spr        =     319
Ymax_Spr        =     255
Start:
                BSR      Init          ;inicjacja procedury
Wait:           CMP.B   #$ff,$dff006  ;czekanie na ramkę
                BNE      Wait
                BSR      Control_Sprite ;kontrolujemy poruszanie się spritea
                BSR      Sprite_Position;zapisujemy nową pozycję spritea do
                                   ;struktury
                BTST     #6,$bfe001
                BNE      Wait
                RTS

;-----
;Kontrola ruchu spritea na ekranie
;-----
Control_Sprite:
                MOVE.W   SprY,D0
                ADD.W     #Wys_Spr,D0
                CMP.W     #Ymax_Spr,D0
                BLE       Ok1
                MOVE.W   #-1,Move_Y
Ok1:            MOVE.W   SprX,D0
                ADD.W     #16,D0
                CMP.W     #Xmax_Spr,D0
                BLE       Ok2
                MOVE.W   #-1,Move_X
Ok2:            CMP.W     #Ymin_Spr,SprY
                BGE       Ok3
                MOVE.W   #1,Move_Y
Ok3:            CMP.W     #Xmin_Spr,SprX
                BGE       Ok4
                MOVE.W   #1,Move_X
Ok4:            MOVE.W   Move_X,D0
                ADD.W     D0,Sprx
                MOVE.W   Move_Y,D0
                ADD.W     D0,Spry
                RTS

;-----
;Sprite Real Coodrs
;D0-X , D1-Y, A0-Sprite
;-----
Sprite_Position:
                MOVE.W   Sprx,D0
                MOVE.W   Spry,D1
                LEA       Sprite,A0
                ADDI.W    #X_Offs_Spr,D0
                ADDI.W    #Y_Offs_Spr,D1
                BTST     #0,D0
                BEQ       No_Second
                ORI.B     #1,3(A0)
                BRA       Second

```

```

No_Second:    BCLR    #0,3(A0)
Second:       ASR.W    #1,D0
              MOVE.B   D0,1(A0)
              CMP.W    #$ff,d1
              BLE      No_Rast_0
              ORI.B    #4,3(A0)
              BRA      Rast_0
No_Rast_0:    BCLR    #2,3(A0)
Rast_0:       MOVE.B   D1,(A0)
              ADD.W    #Wys_Spr,D1
              CMP.W    #$ff,d1
              BLE      No_Rast_1
              ORI.B    #2,3(A0)
              BRA      Rast_1
No_Rast_1:    BCLR    #1,3(A0)
Rast_1:       MOVE.B   D1,2(A0)
              RTS

;-----
;Inicjacja programu
;-----
Init:         LEA      Block,A0
              MOVE.L   #Fikcyjna_Struktura,D0
              MOVE.W   #$0124,d1
              MOVEQ    #13,D2
Loop:         MOVE.W   D1,(A0)+
              ADDQ.W   #2,D1
              SWAP     D0
              MOVE.W   D0,(A0)+
              DBF      D2,Loop
              MOVE.L   #Copper,$dff080
              MOVE.L   #Sprite,D0
              MOVE.W   D0,MemL
              SWAP     D0
              MOVE.W   D0,MemH
              LEA      Planes,A0
              MOVE.L   #Screen,D0
              MOVE.W   D0,6(A0)
              SWAP     D0
              MOVE.W   D0,2(A0)
              RTS

;-----
Copper:       DC.L     $008e2c50
              DC.L     $00902cc1
              DC.L     $00920038
              DC.L     $009400d0
              DC.L     $00968020
              DC.L     $01080000
              DC.L     $010a0000
              DC.L     $01000000
              DC.L     $01020000
              DC.L     $01040000
              DC.L     $01000000
Planes:       DC.L     $00e00000
              DC.L     $00e20000
              DC.L     $01800000
              DC.L     $01820000
              DC.L     $01a20f80
              DC.L     $01a40f00
              DC.L     $01a60ff0
              DC.W     $0120
MemH:         DC.W     $0000
              DC.W     $0122
MemL:         DC.W     $0000
Block:        BLK.L    $e,$0
              DC.L     $01001200
              DC.L     $fffffffe

```

```

Sprite:      DC.B      0,0,0,0
             DC.W      %0000000000000000,%0000000000000000
             DC.W      %0000000100000000,%0000001110000000
             DC.W      %0000000100000000,%0000010001000000
             DC.W      %0000000000000000,%0000100100100000
             DC.W      %0000000000000000,%0001000000010000
             DC.W      %0000000000000000,%0010000000001000
             DC.W      %0000000000000000,%0100000000000100
             DC.W      %0110000000001100,%0101000000010100
             DC.W      %0000000000000000,%0100000000000100
             DC.W      %0000000000000000,%0010000000001000
             DC.W      %0000000000000000,%0001000000010000
             DC.W      %0000000000000000,%0000100100100000
             DC.W      %0000000100000000,%0000010001000000
             DC.W      %0000000100000000,%0000001110000000
             DC.W      %0000000000000000,%0000000000000000
             DC.W      %0000000000000000,%0000000000000000

```

```

Fikcyjna_Struktura:  DC.L      0

```

```

Move_X:      DC.W      1
Move_Y:      DC.W      1

```

```

Sprx:        DC.W      0
Spry:        DC.W      0

```

```

Screen:      BLK.B     40*256,0

```

```
;
```

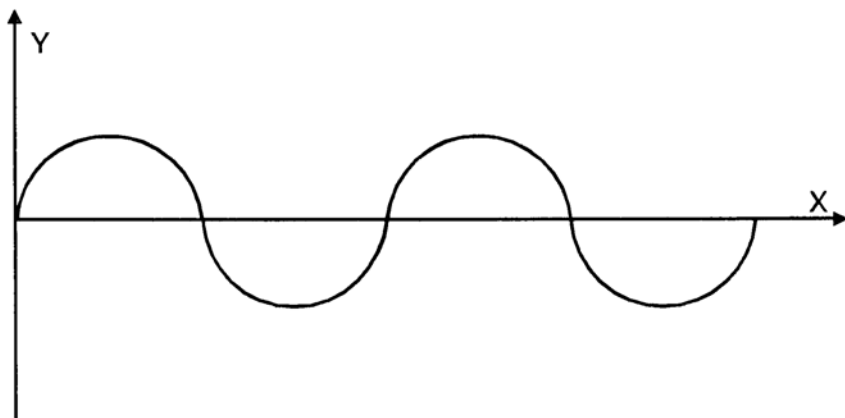
Rozdział VIII

Obsługa dźwięku

8.1 Wstęp

O tym, że Amiga posiada niesamowite, jak na komputer domowy, możliwości odtwarzania dźwięku, nikomu chyba nie trzeba chyba mówić. Możliwości te istnieją dzięki zamontowaniu w niej czterech przetworników cyfrowo-analogowych. Do czego one służą, dowiemy się za chwilę.

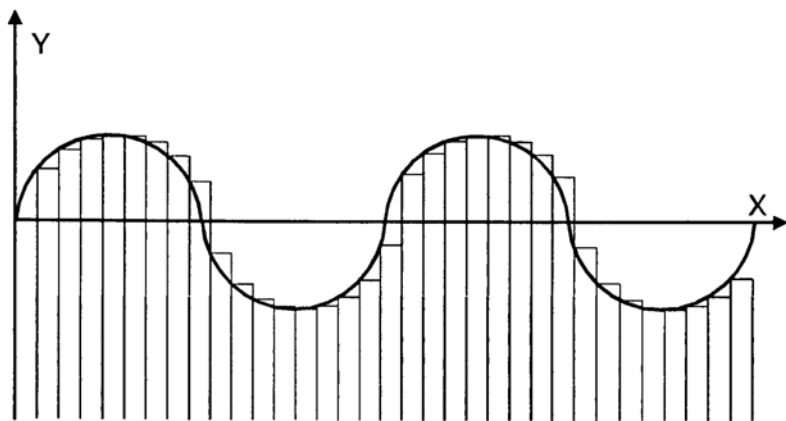
Dźwięk dociera do naszych uszu przez powietrze w postaci powtarzających się cykli zmian ciśnienia w postaci fal dźwiękowych. Natomiast fale można przedstawić za pomocą wykresu... Oto najprostszy przykład fali sinusoidalnej. Oczywiście jest to tylko jej wycinek.



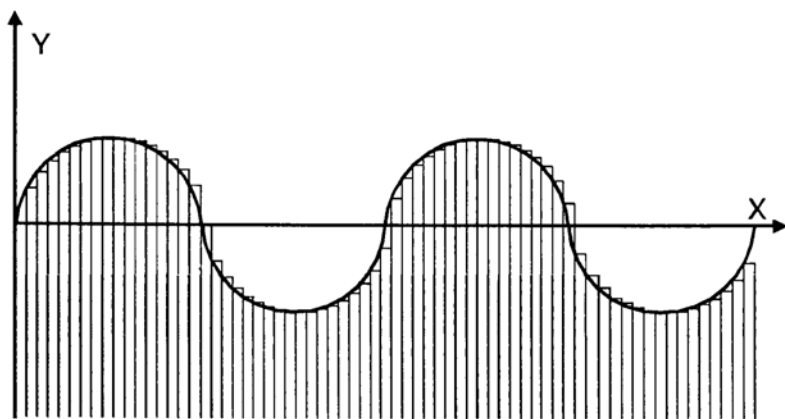
Jeśli wykres fali znajduje się po osi x, to w tym czasie membrana głośnika zostanie ugięta. Jeśli jednak będzie znajdować się ponad osią (przyjmie wartości dodatnie), to membrana zostanie wygięta.

W postaci analogowej sygnał znajduje się np. na płycie gramofonowej czy zwykłej taśmie magnetofonowej. Aby jednak można było zapisać dźwięk do pamięci komputera, należy odpowiednim wartościom natężenia prądu przyporządkować liczby. Dokonujemy tego za pomocą przetwornika analogowo-cyfrowego (A/C), czyli samplera. Proces ten nazywamy digitalizacją. Nie możemy także zapisywać wszystkich danych analogowych (po prostu nie jest to możliwe). Co najwyżej można sprawdzać co jakiś okres czasu, napięcie prądu i dopiero wtedy przypisywać odpowiednią wartość i zapisywać ją do pamięci.

Jak widać, im większa jest częstotliwość próbkowania (czyli sprawdzania natężenia prądu), tym lepsza jest jakość dźwięku, a także większa ilość pamięci potrzebnej do zapamiętania dźwięku. W praktyce wystarcza jednak częstotliwość 11 – 14 kHz.



niska częstotliwość próbkowania



wysoka częstotliwość próbkowania

Amiga używa do zapisu jednej próbki sample (czyli dźwięku zapisanego w postaci cyfrowej) jednego bajtu. Tak więc jednemu sygnałowi analogowemu, przypisana może być wartość od -128 do +127. Tak więc Amiga w porównaniu z odtwarzaczem płyt kompaktowych (który używa 16 bitów na próbkę), wypada nieco słabo. Dzieje się tak, ponieważ dźwięk w Amidze jest odtwarzany z mniejszą dokładnością.

Proces odtwarzania dźwięku jest odwrotny do procesu digitalizacji. Pobrane wartości są zamieniane na impulsy elektryczne za pomocą przetwornika cyfrowo-analogowego (C/A) i odtwarzane przez głośnik.

8.2 Przetworniki C/A

Wcześniej wspomniałem, Amiga posiada 4 przetworniki C/A, ponumerowane od 0 do 3. Przetworniki (od teraz nazywane kanałami), połączone są następująco: 0 i 3 oraz 1 i 2. Połączenie przetworników daje efekt stereo. Kanały 0 i 3 tworzą kanał lewy, a kanały 1 i 2 — prawy.

8.3 Odtwarzanie dźwięku

Skoro już wiemy jak zapamiętywane są dźwięki w pamięci, to możemy się pokusić o ich odtworzenie. Każdy z czterech kanałów ma przyporządkowane 5 rejestrów, które kontrolują odtwarzanie dźwięku za pośrednictwem kanałów DMA.

8.3.1 Określanie adresu sampla w pamięci — AUDxLCH i AUDxLCL

Ta para rejestrów wskazuje kanałom DMA adres początkowy (starsze i młodsze słowo) sampla w pamięci. Blok danych musi znajdować się w pamięci CHIP.

Przypuśćmy, że mamy sampla pod adresem \$23332. Tak więc do AUDxLCH zapisujemy wartość \$0002, a do AUDxLCL \$3332.

Uwaga: rejestry te nie zmieniają swojej zawartości przy pobieraniu przez DMA kolejnych próbek dźwięku do odtworzenia (tak jak to się działo z pointerami bitplanów.) — DMA kopiuje ich zawartości do rejestrów „podręcznych” i w miarę pobierania słów z pamięci, zmienia ich zawartość (oczywiście chodzi o rejestry podręczne).

8.3.2 Określanie długości danych do odtworzenia — AUDxLEN

Rejestr AUDxLEN zawiera długość danych do odtworzenia podaną w słowach. Długość danych musi być parzysta (najmłodszy bit tego rejestru jest ignorowany). Poza tym dane fali nie mogą przekroczyć długości 131070 (\$ffff * 2) bajtów.

Pomimo, że istnieje ograniczenie dla długości fali, to istnieje możliwość odtworzenia dźwięku dłuższego niż 131 070 bajtów. W tym celu należy skorzystać z tego, że Amiga buforuje zawartość rejestrów AUDxLEN i AUDxLC, używając do odtwarzania dźwięku rejestrów zapasowych. Pozwala to na zmianę rejestrów AUDxLC i AUDxLEN, bez wpływu na aktualnie odtwarzany dźwięk. Jeśli zawartość tych rejestrów zostanie zmieniona, zanim aktualny dźwięk zostanie odegrany od nowa, to po zakończeniu odgrywania aktualnej fali dźwiękowej, kanał zacznie odgrywać następną (określoną przez wyżej wymienione rejestry). Dzięki zastosowaniu tej techniki, można bez problemu odtwarzać bardzo długie fragmenty muzyczne.

Należy jednak pamiętać, aby nie zmieniać zawartości tych rejestrów, zanim ich DMA nie skopiowało. Można to sprawdzić kilkoma metodami:

- Zastosowanie programowej pętli opóźniającej. Metoda ta (stosowana zresztą w Protackerze) wymaga, niestety, indywidualnego ustawienia parametrów opóźnienia, dla różnych mikroprocesorów.
- Odczekanie xx linii rastra. Zwykle stosowane jest opóźnienie 5 do 6 linii rastra. Wtedy DMA zdąży pobrać maksymalnie 10 próbek dźwięku i z czystym sumieniem można załadować na nowo w/w rejestry.
- Użycie układów CIA w celu odczekania określonego czasu. Można użyć jednego z timerów (A lub B) i udostępnić przerwanie danego układu CIA. Wtedy przerwanie będzie generowane po określonym czasie. W celu dokładniejszych wskazówek, co do wykorzystania liczników układów CIA, zajrzyj do rozdziału poświęconego kontroli systemu.
- Użycie przerwania *Audio* (poziom 4). Każdy z kanałów dźwiękowych może wygenerować przerwanie 4 poziomu zanim rozpocznie się nowy cykl odtwarzania dźwięku. Gdy przerwanie te się pojawi, wtedy możemy najspokojniej w świecie zmienić zawartość powyższych rejestrów. Jednak należy uważać przy bardzo wysokich częstotliwościach. Wtedy przerwania mogą być generowane za często dla systemu Amigi.

Opisana wyżej cecha buforowania rejestrów AUDxLC i AUDxLEN może być wykorzystana do innych celów. Zwykle, gdy DMA pobierze ostatnią próbkę dźwięku do odtworzenia, to zbuforowana poprzednio zawartość rejestru AUDxLEN wynosi 0 (po każdym pobraniu słowa danych jest zmniejszana o 1), a zbuforowana zawartość AUDxLC wynosi:

$$\text{Wartość początkowa} + 2 \cdot \text{AUDxLEN}$$

(jest zwiększana po każdym pobraniu słowa o 2). Potem DMA znowu buforuje zawartość tych rejestrów i dźwięk jest odtwarzany na nowo, i tak w kółko. Z cechy tej korzystają programy muzyczne, umożliwiając tworzenie „zapętlnych” sampli. Po prostu odtwarzają dźwięk, czekają aż zawartości rejestrów zostaną zbuforowane i zapisują:

- do AUDxLEN długość pętli w słowach,
- do AUDxLC adres początku pętli.

Dzięki temu można tylko raz odegrać dany dźwięk. Wystarczy zapisać do AUDxLEN \$0000 wtedy, gdy DMA zbuforuje zawartość rejestrów.

8.3.3 Ustalanie głośności kanału — AUDxVOL

Aby określić głośność odtwarzanego dźwięku, należy wpisać do rejestru AUDxVOL odpowiednią wartość. Wartość 64 jest maksymalną i dającą największe natężenie dźwięku, a 0 jest wartością minimalną, „najcichszą”.

Należy jednak pamiętać, że dwa dźwięki odtwarzane z takim samym ustawieniem głośności, nie muszą być wcale tak samo słyszalne. Dzieje się tak dlatego, ponieważ natężenie dźwięku zależy także od natężenia kształtu fali.

8.3.4 Określanie tempa odtwarzania dźwięku — AUDxPER

Chcąc odtworzyć dźwięk, musimy określić jego tempo odtwarzania, czyli częstotliwość. Nie jest to jednak ta sama częstotliwość co przy próbkowaniu, lecz okres czasu pomiędzy pobraniem kolejnej próbki dźwięku do odtworzenia. Dokładniej, określamy go podając ilość cykli zegara, jaka powinna upłynąć pomiędzy pobraniem kolejnych danych. Jest to swego rodzaju licznik: DMA po rozpoczęciu odtwarzania dźwięku zmniejsza zawartość AUDxPER o jeden w każdym cyklu zegarowym i dopiero gdy jego wartość osiągnie zero następuje wysłanie następnej próbki.

Kanały DMA mogą pobierać maksymalnie 2 próbki danych (2 bajty) podczas wyświetlania jednej linii poziomej ekranu (jednego rastra). Więc maksymalna ilość próbek do odtworzenia w ciągu jednej sekundy wynosi:

$$\left(2 \frac{\text{próbki}}{\text{raster}}\right) \cdot (312 \text{ rastrów}) \cdot \left(50 \frac{\text{ramek}}{\text{s}}\right) = 31200 \text{ próbek}$$

Jednak wartość ta jest czysto teoretyczna. Istnieją pewne opóźnienia, natury niedoskonałości sprzętowej, więc faktycznie maksymalna wartość wynosi 28867 próbek/s.

Oto przykład: chcemy odtworzyć sample o długości z częstotliwością 10 kHz. Każda próbka zostanie odtworzona w ciągu 1/10000 s., czyli mamy 100 µs na próbkę. Ponieważ cykl trwa 0.279365 µs, zatem do rejestru AUDxPER wpisujemy $100/0.279365 = 355$ cykli/próbkę. Zatem im niższa jest wartość okresu, tym większa jest częstotliwość odtwarzania dźwięku. Oto wzór na obliczenie wartości dla rejestru AUDxPER:

$$\text{Okres} = \frac{\text{czas potrzebny na jedną próbkę}}{\text{czas trwania taktu zegarowego}}$$

Tak więc najmniejszą możliwą wartością rejestru AUDxPER do zaakceptowania przez Amigę jest 124. Zaś największa wartość może wynosić 65535, ponieważ rejestr ten jest 16-bitowy.

Jednak często potrzebne jest aby kolejne próbki dźwięku zostały wysyłane z taką samą częstotliwością, z jaką zostały zdigitalizowane. Tak więc jeśli wiesz z jaką częstotliwością digitalizowałeś dźwięk, to użyj następującego wzoru:

$$\text{Okres} = \frac{3579546}{\text{częstotliwość}}$$

Oto przykład: chcemy odtworzyć dźwięk zsamlpowany z częstotliwością 12 kHz.

$$\text{Okres} = \frac{3579546}{120000} = 298$$

Skąd wzięło się to równanie? Otóż, jeśli Amiga zmniejsza rejestr okresu co 0.279365 μ s, to co sekundę będzie następowało zmniejszenie o:

$$\frac{1}{0.000000279365} = 3579546$$

8.3.5 Odtwarzanie dźwięku

Jeśli ustalimy zawartość wszystkich rejestrów, to odtwarzanie dźwięku rozpocznie się dopiero po włączeniu odpowiedniego kanału DMA (odpowiada za to rejestr DMACON). Oto interesujące nas bity w DMACON:

- bit 15 (SET/CLR) — określa, czy ustawione bity mają być zerowane czy ustawiane,
- bit 9 (DMAEN) — włączenie tego bitu uaktywnia kanały DMA,
- bit 3 (AUD3EN) — włączenie kanału DMA dla 3 kanału dźwiękowego,
- bit 2 (AUD2EN) — włączenie kanału DMA dla 2 kanału dźwiękowego,
- bit 1 (AUD1EN) — włączenie kanału DMA dla 1 kanału dźwiękowego,
- bit 0 (AUD0EN) — włączenie kanału DMA dla 0 kanału dźwiękowego.

I jeszcze raz przypomnę znaczenie bitu SET/CLR. Jeśli będziemy chcieli ustawić bity 9 i 0, to wpisujemy do rejestru taką wartość: %1000001000000001 (ustawione bity 0, 9 i SET/CLR). Jeśli będziemy chcieli te bity wyzerować, to wpisujemy do rejestru wartość: %0000001000000001 (ustawione bity 0, 9, a SET/CLR wyzerowany).

8.4 Modulacja dźwięku

Amiga może używać jednego kanału do modulacji drugiego. Modulacja może być częstotliwościowa (zmiana okresu) lub amplitudowa (dotycząca głośności dźwięku). Modulowane mogą być jednocześnie obydwie te parametry, jak i każdy z osobna. Kanał może modulować jedynie kanał o numerze większym. Oznacza to, że kanał 0 nie może być modulowany, a kanał 3 nie może być modulującym.

Za kontrolę modulacji dźwięku odpowiada rejestr ADKCON (\$dff09e), w którym interesują nas następujące bity:

- bit 15 (SET/CLR) — określa czy ustawione bity mają być zerowane czy ustawiane,
- bit 3 (USE3PN) — kanał 3 niczego nie moduluje,
- bit 6 (USE2P3) — kanał 2 moduluje okres kanału 3,
- bit 5 (USE1P2) — kanał 1 moduluje okres kanału 2,
- bit 4 (USE0P1) — kanał 0 moduluje okres kanału 1,
- bit 3 (USE3VN) — kanał 3 niczego nie moduluje,

- bit 2 (USE2V3) — kanał 2 moduluje głośność kanału 3,
- bit 1 (USE1V2) — kanał 1 moduluje głośność kanału 2,
- bit 0 (USE0V1) — kanał 0 moduluje głośność kanału 1.

Przy korzystaniu z modulacji dane kanału modulującego nie są traktowane jak 8-bitowe próbki, lecz są łączone w 16-bitowe słowa. W przypadku modulacji okresu, otrzymany wynik zapisywany jest do rejestru AUDxPER. Natomiast gdy będziemy modulować głośność, wynik zostanie zapisany do rejestru AUDxVOL.

W wypadku gdy będziemy modulować obydwie te wartości naraz, to pobierane słowa będą traktowane naprzemiennie: głośność, okres, głośność, okres.

Pomimo tego, że modulacja dźwięku może dawać dość ciekawe rezultaty, to w praktyce jest ona stosowana bardzo rzadko. Dzieje się tak dlatego, ponieważ kanał modulujący nie może być jednocześnie użyty do odtwarzania dźwięku.

Modulację dźwięku na Amidze tworzy się głównie programowo. Daje to większe możliwości, a także wszystkie cztery kanały mogą być na raz używane do odtwarzania dźwięku.

8.5 Filtr dolnoprzepustowy

Amiga posiada także wbudowany jeden filtr dźwiękowy. Do czego on służy? Otóż przy digitalizacji dźwięku dochodzi problem szumów. Aby temu zaradzić, Amigę wyposażono w filtr dolnoprzepustowy o częstotliwości granicznej rzędu 7kHz. W wyniku tego wysokie częstotliwości (wyższe od częstotliwości granicznej) zostają „obcięte”. Właśnie one są odpowiedzialne za powstawanie szumów. Filtr ten jest jednak dość rzadko stosowany, ponieważ obejmuje swoim działaniem jednocześnie wszystkie cztery kanały i dźwięk jest nieco przytłumiony.

Włączenie filtra dolnoprzepustowego sygnalizowane jest przyciemnieniem diody POWER. Włącza się go poprzez ustawienie bitu #1 w rejestrze \$bfe001.

8.6 Tryb „ręczny”

Prawie wszystko co dotychczas omówiliśmy, dotyczyło automatycznego pobierania kolejnych próbek przez kanały DMA. Istnieje jednak metoda, by mikroprocesor pobierał dane do odtwarzania i wysyłał je do przetworników C/A.

Amiga posiada rejestry AUDxDAT do których DMA automatycznie ładuje pobrane próbki dźwięku. Inaczej mówiąc jest to bufor danych dla danego kanału dźwiękowego. Aby z niego skorzystać, należy najpierw wyłączyć kanały DMA Audio. Potem można wpisywać kolejne wartości do tego rejestru (najpierw wysyłany do odtworzenia jest straszys, a potem młodszy bajt).

Aby była zachowana odpowiednia synchronizacja, to za każdym razem, gdy rejestr jest przygotowany na przyjęcie danych, pojawia się przerwanie 4 poziomu.

Ten sposób można wspaniale wykorzystać do „powiększenia” liczby kanałów dźwiękowych do np. 8 czy 16. Co prawda, pogarsza się wtedy nieco jakość, ale efekt może być wspaniały (w zależności od procedury odgrywanej muzykę). Po prostu procesor pobiera próbki z dwóch „fikcyjnych” kanałów i oblicza ich średnią. Potem je ewentualnie przetwarza i wysyła do odtworzenia. Procedura musi być oczywiście odpowiednio szybka, by zdążyła wykonać tyle operacji w dość krótkim czasie.

Metoda ta jednak jest dość czasochłonna i niezbyt często używana, ponieważ procesor główny jest wtedy zajęty odtwarzaniem dźwięku.

8.7 Wykorzystanie dźwięku w praktyce

W części praktycznej tego rozdziału ograniczymy się tylko do odtwarzania sampli. Na początek podaję adresy poszczególnych rejestrów kanałów dźwiękowych:

kanal 0:

AUD0PTH — \$dff0a0,
AUD0PTL — \$dff0a2,
AUD0LEN — \$dff0a4,
AUD0PER — \$dff0a6,
AUD0VOL — \$dff0a8,
AUD0DAT — \$dff0aa — używany w trybie ręcznym lub przez DMA,

kanal 1:

AUD1PTH — \$dff0b0,
AUD1PTL — \$dff0b2,
AUD1LEN — \$dff0b4,
AUD1PER — \$dff0b6,
AUD1VOL — \$dff0b8,
AUD1DAT — \$dff0ba — używany w trybie ręcznym lub przez DMA,

kanal 2:

AUD2PTH — \$dff0c0,
AUD2PTL — \$dff0c2,
AUD2LEN — \$dff0c4,
AUD2PER — \$dff0c6,
AUD2VOL — \$dff0c8,
AUD2DAT — \$dff0ca — używany w trybie ręcznym lub przez DMA,

kanal 3:

AUD3PTH — \$dff0d0,
AUD3PTL — \$dff0d2,
AUD3LEN — \$dff0d4,

AUD3PER — \$dff0d6,
 AUD3VOL — \$dff0d8,
 AUD3DAT — \$dff0da — używany w trybie ręcznym lub przez DMA.

Poniżej przedstawione są 2 procedury wykorzystujące poznane w tym rozdziale wiadomości. Pierwsza odgrywa pojedynczy dźwięk nagrany na dysku jako plik SOUND, wykorzystując cechę buforowania rejestrów AUDxLC i AUDxLEN. Jeśli nie posiadasz żadnego sampla, to dla eksperymentu możesz doczytać dowolny plik (np. LIBS/ICON.LIBRARY, itp.). Procedura **Modulation** za pomocą kanału 1 odgrywa falę sinusoidalną, modulując przy tym jej głośność za pomocą kanału 0.

```

AUD0LCN      =      $a0
AUD0LEN      =      $a4
AUD0PER      =      $a6
AUD0VOL      =      $a8
AUD0DAT      =      $aa

AUD1LCN      =      $b0
AUD1LEN      =      $b4
AUD1PER      =      $b6
AUD1VOL      =      $b8
AUD1DAT      =      $ba

DMACON       =      $96
ADKCON       =      $9e

VHPOSR       =      $06

;-----
PlaySample:
    LEA        $dff000,a6
    MOVE.L     #Sample,AUD0LCN(A6)
    MOVE.W     #[SampleEnd-Sample]/2,AUD0LEN(A6)
    MOVE.W     #31,AUD0PER(A6)
    MOVE.W     #30,AUD0VOL(A6)
    MOVE.W     #$8201,dmacon(a6)

Loop:        MOVEQ    #6-1,D0                ; czekamy 6 linii rastra
Wait:        MOVE.B   VHPOSR(A6),D1           ; aby DMA zbuforowało zawartość
            CMP.B     VHPOSR(A6),D1           ; rejestrów AUDxLC i AUDxLEN
            BEQ.S     Wait
            DBF       D0,Loop

            MOVE.L     #SampleEnd,AUD0LCN(A6) ; po odtworzeniu sampla do końca
            MOVE.W     #2,AUD0LEN(A6)         ; odtwarzany będzie "pusty" dźwięk

Wait_Mouse:
    BTST       #6,$bfe001                    ; oczekiwanie na myszkę
    BNE.S      Wait_Mouse
    MOVE.W     #$0001,dmacon(a6)
    RTS

Sample:      Incbin   "df0:sound"
SampleEnd:   DC.L     0

;-----

Modulation:
    LEA        $dff000,a6
    MOVE.L     #Sample,AUD1LCN(A6)
    MOVE.W     #[SampleEnd-Sample]/2,AUD1LEN(A6)
    MOVE.W     #10,AUD1PER(A6)                ; ustawienie zawartości
    MOVE.W     #20,AUD1VOL(A6)                ; rejestrów dla kanału modulowanego

```

```

        MOVE.L    #Modulation,AUD0LCN(A6)
        MOVE.W    #[ModEnd-Modulation]/2,AUD0LEN(A6)
        MOVE.W    #$ffff,aud0per(a6)           ;ustawienie zawartości
        MOVE.W    #0,AUD0VOL(A6)               ;rejestrów dla kanału modulującego
        MOVE.W    #$8001,adkcon(a6)            ;kanał 0 moduluje głośność kanału 1
        MOVE.W    #$8203,dmacon(a6)

Mouse_Wait:
        BTST      #6,$bfe001
        BNE.S     Mouse_Wait
        MOVE.W    #$0001,adkcon(a6)
        MOVE.W    #$0003,dmacon(a6)
        RTS

Sample:      DC.B    -128,-124,-119,-113,-107,-100,-91,-82,-69
              DC.B    -55,-38,-20,-9,0,2,5,11,19,26,39,56,76,83
              DC.B    92,104,117,127,120,111,104,93,85,73,65,59
              DC.B    45,33,20,14,9,3,-8,-15,-24,-35,-43,-59
              DC.B    -83,-100,-122

SampleEnd:   Even                                         ;wyrównanie do parzystego adresu

Modulation:  DC.W    0,0,0,0,0,2,4,7,9,12,15,19,23,28,33,37,42
              DC.W    48,54,60,64,64,64,64,64

ModEnd:
;-----

```

8.8 Odtwarzanie modułów muzycznych

Aby odtworzyć moduł muzyczny napisany za pomocą programu typu „tracker”, należy najpierw zaopatrzyć się w odpowiednią procedurę odgrywającą. Są one z reguły rozpowszechniane na dyskietkach razem z programem muzycznym. Jeśli masz już dowolną „trackerowską odgrywaczkę” to czas na poznanie ogólnych zasad ich używania.

- Inicjacja procedury odgrywającej — zwykle na początku programu musimy wywołać etykietę **mn_INIT**. **mn** oznacza w tym wypadku przedrostek etykiet procedury odgrywającej. W przypadku procedury ProRunera będzie to **PR** a Protrackera **MT**. Oto jak możemy wywołać procedurę inicjującą:

- procedura odtwarzająca muzykę z Protrackera

```
JSR      Mt_Init
```

- procedura odtwarzająca muzykę z Noisetrackera

```
JSR      Nt_Init
```

- procedura odtwarzająca muzykę z ProRunera

```
JSR      Pr_Init....
```

- Wywoływanie głównej procedury odtwarzającej — każda procedura odgrywająca wymaga wywołania jej co każdą ramkę. Przy pisaniu dema całodyskowego, najlepiej będzie umieścić muzykę na przerwaniu VBLANK. Zwykle procedura taka nazywa się **mn_MUSIC** lub **mn_PLAY**. Oto sposób wykorzystania:

```

      .
      .
      .
Beam_Loop:  CMP.B  #$ff,$dff006    ;czekanie na ramkę
            BNE   Beam_Loop
            JSR   Mt_Music        ;skocz do procedury odgrywającej
                                   ;muzykę
            BTST  #6,$bfe001      ;czy lewy przycisk myszki
                                   ; przyciśnięty?
            BNE   Beam_Loop       ;jeśli nie to skocz do Beam_Loop
      .
      .
      .

```

- Zakończenie odgrywania — odbywa się zwykle po zakończeniu skoków do mn_Music i wywołaniu etykiety mn_END.

Oto kompletny przykład wykorzystania procedury odgrywającej, zawartej na dysku z Protrackerem.

```

;-----
            JSR   Mt_Init          ;inicjujemy procedurę odgrywającą
Beam_Loop:  CMP.B  #$ff,$dff006    ;czekanie na ramkę
            BNE   Beam_Loop
            JSR   Mt_Music        ;skocz do procedury odgrywającej muzykę
            BTST  #6,$bfe001
            BNE   Beam_Loop
            JSR   Mt_End          ;kończymy odgrywanie muzyki
            RTS
;-----

```


Rozdział IX

Kontrola systemu i obsługa interfejsów

9.1 Wprowadzenie

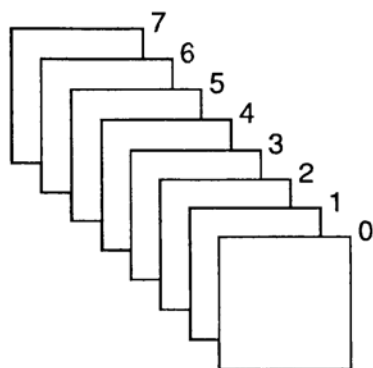
Obsługa samych układów specjalizowanych nie wystarcza. Bardzo często musimy wiedzieć, jak wykryć kolizje pomiędzy obiektami, co to są przerwania i jak je wykorzystać, jak odczytywać i zapisywać dane na dyskietce, a także co zrobić aby nasze programy działały poprawnie z systemem operacyjnym.

9.2 Priorytety wizji

Aby uzyskać efekty przestrzenne, obiekty znajdujące się bliżej, powinny przestaniać obiekty znajdujący się nieco dalej. W przypadku tworzenia rysunku nie ma problemu — po prostu rysujemy od tyłu, wtedy obiekty znajdujące się bliżej samoczynnie zasłaniają obiekty dalsze. Sprawa się jednak komplikuje gdy używamy spritey i playfieldy.

9.2.1 Priorytety spriteów

Na ekranie można wyświetlić do ośmiu spriteów o numerach od 0 do 7. Na ekranie są wyświetlane w kolejności numerów — od najwyższego do najniższego. Z tego względu priorytety spriteów są nie zmienialne, a spritey o wyższych numerach, mają niższy priorytet wyświetlania. Oto schematyczny rysunek:



9.2.2 Grupowanie sprite'ów

Spritey są grupowane po dwa, w dwóch przypadkach:

- w celu wykrycia kolizji,
- dla priorytetów playfieldów.

Spritey są połączone w następujące grupy:

- sprite 0 ze spritem 1,
- sprite 2 ze spritem 3,
- sprite 4 ze spritem 5,
- sprite 6 ze spritem 7.

9.2.3 Ustawianie priorytetów pomiędzy spriteami a playfieldem

Priorytety wyświetlania playfieldów (tryb DPM) umiemy już zmieniać (patrz rozdział 5.4.3). Teraz jednak zajmiemy się priorytetami wyświetlania sprite'ów i playfieldów.

Bity 0 – 5 w rejestrze BPLCON2 odpowiadają za umieszczenie sprite'ów względem playfieldów. Przypomnijmy sobie dokładne znaczenie bitów w tym rejestrze.

BPLCON2:

bity 15 – 7 — nie używane (ustawić na 0),

bit 6 — PF2PRI,

bity 5 – 3 — PF2P2 – PF2P0,

bity 2 – 0 — PF1P2 – PF1P0.

Przy wyłączonym trybie DPM, bity 5 – 3 są priorytetami dla normalnego playfieldu.

A oto tabelka ustawienia bitów dla pożądanых priorytetów. PFx jest tutaj playfieldem, a SP01 — grupą sprite'ów 0 i 1.

Ustawienie bitów PFxP2 – PFxP0	Priorytety (do najmniejszego)				
000	PFx	SP01	SP23	SP45	SP67
001	SP01	PFx	SP23	SP45	SP67
010	SP01	SP23	PFx	SP45	SP67
011	SP01	SP23	SP45	PFx	SP67
100	SP01	SP23	SP45	SP67	PFx

9.3 Wykrywanie kolizji

W grach bardzo często konieczne jest sprawdzanie kolizji pomiędzy bohaterem, a jakimś stworkiem. Można to zrobić „ręcznie”, porównując współrzędne obu obiektów lub sprzętowo.

Amiga pozwala na wykrycie kolizji na poziomie sprzętowym pomiędzy grupami spriteów lub playfieldami w dowolnej kombinacji. Możemy więc wykryć kolizje pomiędzy dwoma grupami spriteów, grupą spriteów, a playfieldem oraz dwoma playfieldami.

Kolizje wykrywamy odczytując rejestr CLXDAT — należy pamiętać, że jest on kasowany zaraz po odczycie. Oto znaczenie poszczególnych bitów w tym rejestrze:

Nr bitu	kolizja	Nr bitu	kolizja
15	nie używany	7	bitplany parzyste i sprite 4 (lub 5)
14	sprite 4 (lub 5) i sprite 6 (lub 7)	6	bitplany parzyste i sprite 2 (lub 3)
13	sprite 2 (lub 3) i sprite 6 (lub 7)	5	bitplany parzyste i sprite 0 (lub 1)
12	sprite 2 (lub 3) i sprite 4 (lub 5)	4	bitplany nieparzyste i sprite 6 (lub 7)
11	sprite 0 (lub 1) i sprite 6 (lub 7)	3	bitplany nieparzyste i sprite 4 (lub 5)
10	sprite 0 (lub 1) i sprite 4 (lub 5)	2	bitplany nieparzyste i sprite 2 (lub 3)
9	sprite 0 (lub 1) i sprite 2 (lub 3)	1	bitplany nieparzyste i sprite 0 (lub 1)
8	bitplany parzyste i sprite 6 (lub 7)	0	bitplany parzyste i nieparzyste

Spritey, których numery podane są w nawiasach, nie są normalnie uwzględniane w kolizji. Możemy o tym jednak decydować, zmieniając zawartość bitu CLXCON. Należy także zauważyć, że nie ma tutaj podziału na playfieldy. Kolizje są uwzględniane nawet dla wyłączonego trybu DPM.

A oto opis rejestru CLXCON, który odpowiada za różne cechy wykrywania kolizji.

Nr bitu	funkcja	Nr bitu	funkcja
15	wykrywaj kolizję spritea 7	7	wykrywaj kolizję bitplanu 2
14	wykrywaj kolizję spritea 5	6	wykrywaj kolizję bitplanu 1
13	wykrywaj kolizję spritea 3	5	wartość bitu kolizji bitplanu 6
12	wykrywaj kolizję spritea 1	4	wartość bitu kolizji bitplanu 5
11	wykrywaj kolizję bitplanu 6	3	wartość bitu kolizji bitplanu 4
10	wykrywaj kolizję bitplanu 5	2	wartość bitu kolizji bitplanu 3
9	wykrywaj kolizję bitplanu 4	1	wartość bitu kolizji bitplanu 2
8	wykrywaj kolizję bitplanu 3	0	wartość bitu kolizji bitplanu 1

Bity 5 – 0 określają wartość bitu, jaka ma być zgłoszona w wypadku kolizji danego bitplanu. Natomiast bity 15 – 12 określają, czy w danej parze sprite'ów ma być sprawdzana kolizja sprite'a o numerze nieparzystym.

9.4 Pozycja promienia, jej sprawdzanie i związek z animacją

Chcąc zsynchronizować działanie programu wykonywanego przez mikroprocesor z promieniem wizji (żeby zapewnić płynną animację), można kontrolować pozycję promienia i to na kilka sposobów. Pierwszym z nich jest odczytanie zawartości rejestrów VPOSR i VHPOSR. Można także zapisać nową pozycję promienia za pomocą rejestrów VPOSW i VHPOSW. Drugą metodą jest wykorzystanie przerwań (omówione w następnym rozdziale). Układając odpowiednią copperlistę i umieszczając w niej zgłoszenie przerwania, możemy co ramkę zgłaszać Motoroli o zmianie (lub zmianach) pozycji promienia wizji. Trzecią metodą, także związaną z przerwaniami, jest wykorzystanie przerwania VBLANK. Mikroprocesor dostaje wtedy zgłoszenie o wygaszaniu pionowym.

A oto znaczenie rejestrów VPOSR i VHPOSR (VPOSW i VHPOSW pozwalają na zmianę aktualnej pozycji promienia, a VPOSR i VHPOSR pozwalają tylko na jej odczyt):

- VPOSR — adres \$dff004
- bit 15 — (LOF) Używany w trybie interlace. Wskazuje gdy wystąpi tzw. *długa ramka* (LOng Frame),
 - bity 14 – 1 — nie używane,
 - bit 0 — najstarszy bit pozycji pionowej.
- VHPOSR — adres \$dff006
- bity 15 – 8 — niższe bity pozycji pionowej,
 - bity 7 – 0 — bity H8 – H1 pozycji poziomej.

Można zauważyć, że ignorowany jest najmłodszy bit pozycji poziomej (H0), przez co następna pozycja pozioma może być większa od poprzedniej o 2 punkty w poziomie.

Teraz kilka przykładów na odczytywanie pozycji promienia.

```
Wait_Raster:  MOVE.L  $dff004,d0
              LSR.L  #8,D0
              ANDI.W  #$1ff,d0
              CMP.W   #$12d,d0
              BNE     Wait_Raster
              RTS
```

Jak widać, przykład ten zezwala nam na czekanie na dowolną linię. Aktualnie ustawiona jest linia \$12d, ale możemy odczekać na każdą inną (lecz nie większą od \$138).

Można to zrobić prościej, ale za to mniej elegancko:

```
Wait_Raster:    CMP.B    #$ff,$dff006
                BNE      Wait_Raster
                RTS
```

Ten przykład jest mało uniwersalny, ale działa. Możemy w nim czekać na linię w zakresie od \$0 do \$ff.

```
Wait_Raster:    CMP.B    #$ff,$dff006
                BNE      Wait_Raster
Wsl:            CMP.B    #$2d,$dff006
                BNE      Wsl
                RTS
```

A to odmiana poprzedniego przykładu. Czekamy na linię poniżej \$ff (w tym wypadku \$ff+\$2d). Amiga w systemie PAL ma dostępnych 138 pozycji pionowych. Po przejściu promienia wizji przez 138 linię, następuje okres wygaszania pionowego i promień powraca do linii \$0. Teraz pomyślmy co nastąpi, gdy napiszemy jakąś procedurę, która będzie operowała na grafice. Niech będzie to np. grafika przestrzenna.

Jeśli napiszemy szybką procedurę, która wszystko obliczy i narysuje, zanim promień wizji osiągnie najniższą linię (\$138), to możemy wyświetlać grafikę z maksymalną częstotliwością (50 obrazów na sekundę, czyli „co ramkę”). Jeśli jednak procedura będzie wolna i nie zmieści się w jednej ramce, to częstotliwość wyświetlania nowych faz animacji będzie odpowiednio niższa. Widać to bardzo często w starszych demach, gdzie animacja wektorowa jest skokowa.

Należy pamiętać, aby *zawsze* przy animacji odczekiwać na jakąś pozycję promienia. Tylko wtedy animacja będzie prawidłowo wyświetlana.

Z wyświetlaniem animacji wiąże się jeszcze jedna rzecz. Co się stanie, gdy będziemy rysować grafikę przestrzenną na ekranie? Pamiętajmy o tym, że cały czas komputer wyświetla ten sam obszar pamięci, w którym rysujemy obiekt.

A oto jakie wykonujemy kolejne kroki:

- czekaj na ramkę,
- wykonaj obliczenia,
- wyczyść ekranu,
- narysuj obiekt,
- skocz do pierwszego kroku.

Gdy teraz będziemy animować kilkanaście kreseczek, to w pewnym momencie okaże się, że któreś kreski zostają narysowane w tym miejscu ekranu, który promień wizji już wyświetlił (a kresceczek tam jeszcze nie było). Dzieje się tak co każdą ramkę i w rezultacie, linie albo nie są widoczne, albo są ucięte. Z kolei gdy animacja będzie następować co kilka ramek, to widoczne będą kolejne fazy rysowania obiektu.

Aby temu zapobiec, przy animacji stosuje się *podwójne buforowanie ekranu*. Jego podstawą jest używanie dwóch ekranów, z których jeden jest wyświetlany, a w międzyczasie na drugim rysowana jest kolejna faza animacji.

Oto kolejne kroki:

- wyświetl zawartość ekranu 1,
- na ekranie 2 rysuj fazę animacji,
- czekaj na ramkę,
- wyświetl zawartość ekranu 2,
- na ekranie 1 rysuj fazę animacji,
- czekaj na ramkę,
- skocz do pierwszego kroku.

Jak widać, przy animacji widoczny będzie ten ekran, na którym aktualnie nic nie rysujemy. Chroni to nas przed opisanymi wcześniej, dodatkowymi efektami.

9.5 Przerwania i stany wyjątkowe

W poprzednich rozdziałach wspominałem już o przerwaniach i stanach wyjątkowych. Czym one są? Wyobraź sobie taką sytuację: Czytasz książkę. Nagle słyszysz dzwonek telefonu. Odkładasz książkę i podnosisz słuchawkę. Po skończonej rozmowie powracasz do czytania i zaczynasz od tego miejsca, w którym przerwałeś. Wynika z tego, że przerwanie jest pewnym wydarzeniem, impulsem, pod wpływem którego komputer (a raczej mikroprocesor) przerywa normalne wykonywanie programu i wykonuje procedurę obsługującą przerwanie. Po zakończeniu powraca do poprzedniego programu i kontynuuje jego wykonywanie od przerwano miejsca. Przerwania są zgłaszane mikroprocesorowi przez inne układy (np. coppera, czy audio) za pomocą zmiany bitów w odpowiednich rejestrach. Tym jednak zajmiemy się później.

9.5.1 Stany wyjątkowe i wektory

Przerwania są odmianą stanów wyjątkowych. Stany wyjątkowe różnią się tym, że są generowane nie z zewnątrz (jak przerwania), lecz przez sam mikroprocesor.

Wyróżniamy następujące typy stanów wyjątkowych (generowanych wewnętrznie):

- błędy wewnętrzne,
- pułapki,
- śledzenie.

Z każdym stanem wyjątkowym i przerwaniem, jest związane w pamięci jedno długie słowo, zwane wektorem. Wektorów jest 256. Wskazują one, gdzie mikroprocesor ma „skoczyć” przy danym stanie wyjątkowym. Motorola 68000 rezerwuje na ten cel adresy od \$00000000 do \$000000400. Jednak nowsze jej wersje (na przykład MC 68020, 68030 czy 68040) posiadają tzw. **VECBASE**. W wyniku tego początek tablicy wektorów możemy przenieść praktycznie w każde miejsce pamięci.

Oto tablica adresów wektorów. Jako ADRES w wypadku procesorów nowszych niż 68000 należy rozumieć przesunięcie względem adresu wskazywanego przez VECBASE.

Numer wektora	Adres	Funkcja
0	000	Wartość początkowa SSP
1	004	Wartość początkowa PC
2	008	Błąd magistrali
3	00C	Błąd adresu
4	010	Nieistniejąca instrukcja
5	014	Dzielenie przez zero
6	018	Instrukcja CHK
7	01C	Instrukcja TRAPV
8	020	Narzucenie uprzywilejowania
9	024	Śledzenie
10	028	Emulator linii A (1010)
11	02C	Emulator linii F (1111)
12 – 14	030 – 038	Nieoznaczone, zarezerwowane
15	03C	Niezainicjowany wektor przerwania
16 – 23	040 – 05C	Nieoznaczone, zarezerwowane
24	060	Fałszywe przerwanie
25	064	Przerwanie (poziom 1)
26	068	Przerwanie (poziom 2)
27	06C	Przerwanie (poziom 3)
28	070	Przerwanie (poziom 4)
29	074	Przerwanie (poziom 5)
30	078	Przerwanie (poziom 6)
31	07C	Przerwanie (poziom 7)
32 – 47	080 – 0BC	Instrukcje TRAP #0 do TRAP #15
48 – 63	0C0 – 0FC	Nieoznaczone, zarezerwowane
64 – 225	100 – 3FF	Wektory przerwania zdefiniowane przez użytkownika

Po przejrzaniu powyższej tablicy można się domyślić, że system operacyjny *Amigi* zapisuje do niektórych wektorów (na przykład 5) adresy wskazujące procedury GURU MEDITATION lub też SOFTWARE FAILURE. Skutkiem tego jest mrugająca, czerwona ramka ze stosownym komunikatem.

W wypadku rozpoznania przez Motorolę stanu wyjątkowego, podejmowane są następujące czynności:

- Na stosie zapisana zostaje wartość PC, oraz zawartość rejestru statusowego.
- Bit T rejestru statusowego zostaje wyzerowany, oraz ustawiony zostaje bit S (bit nadzorca). Celem tej operacji jest zapobiegnięcie sytuacji śledzenia procedury obsługi stanu wyjątkowego. W wypadku obsługi przerwania, uaktualniona zostaje także maska przerwania.
- W razie wystąpienia błędu magistrali lub błędu adresowania, tworzona jest ramka stosu, na której zapisywane są dodatkowe informacje.
- Do licznika programu (PC) załadowany zostaje odpowiedni wektor stanu wyjątkowego, co powoduje wykonanie procedury obsługi tego stanu.

Aby powyższe czynności stały się bardziej zrozumiałe, musimy poznać znaczenie bitów w bajcie systemowym, a także kilka innych rzeczy.

- Bit 15 (T) — bit ten zwany jest bitem śledzenia. Jeśli będzie on ustawiony, to po zakończeniu każdej instrukcji wywołany zostanie stan wyjątkowy określony wektorem numer 9.
- Bit 13 (S) — bit ten kontroluje dostęp mikroprocesora do niektórych instrukcji. Jeśli jest on ustawiony, to procesor znajduje się w trybie nadzorca. Tryb ten umożliwia wykonywanie tzw. instrukcji uprzywilejowanych i zapewnia dostęp do rejestru statusowego. Jeśli jednak bit S będzie wyłączony, to procesor jest w trybie użytkownika. W nim dostęp do rejestru statusowego i instrukcji uprzywilejowanych jest zabroniony. W wypadku narzucenia uprzywilejowania (próby wykonania rozkazu uprzywilejowanego), wykonana zostaje procedura obsługi stanu wyjątkowego nr 8.
- Bity od 10 do 8 zwane są maską przerwania. O tym jednak powiem nieco później.

9.5.2 Przykłady i objaśnienia stanów wyjątkowych

W tym rozdziale podam przykłady inicjacji stanów wyjątkowych i ich wywołania. Należy jednak pamiętać, by nie uruchamiać tych przykładów bezpośrednio, tylko nagrać jako plik wykonywalny (object) i uruchamiać spod CLI. Jeśli jednak chcemy już sprawdzić działanie naszej procedury, to postępujemy tak:

- zachowujemy starą zawartość danego wektora,
- wpisujemy tam adres naszej procedury,
- wywołujemy stan wyjątkowy,
- zapisujemy zachowaną wartość do wektora.

Według takiego schematu wykonywane są wszystkie przykłady zawarte w tym rozdziale.

Uwaga: nie objaśniam wszystkich stanów wyjątkowych, lecz tylko część z nich. Robię tak, ponieważ nie ma żadnego sensu tłumaczenie Tobie jako początkującemu koderowi rzeczy, które mogą być wykorzystane dopiero przy pisaniu własnego assemblera, czy systemu operacyjnego. Jednak zainteresowanych odsyłam do książki pt. „68000 Microprocessor Handbook”, Wiliama Cramera i Gerryego Kanea.

9.5.2.1 Instrukcje nielegalne i niezaimplementowane

Pierwszy przykład zapisuje nowy adres procedury obsługi stanu wyjątkowego dla nielegalnej instrukcji. Stan taki zostaje wywołany w wypadku, gdy procesor pobierze słowo, które nie jest rozpoznawane jako istniejąca instrukcja.

Instrukcje nielegalne to instrukcje:

- posiadające nielegalny kod różny od kodu \$axxx lub \$fxxx,
- używające nielegalnego trybu adresowania,
- błędnie określające rozmiar instrukcji.

Co prawda program assemblera nie zassembleruje żadnej instrukcji tego typu, niemniej wskutek błędów, program może się wydostać spod kontroli i zacząć „latać” po pamięci, gdzie jako program mogą być potraktowane np. dane dla grafiki.

Oprócz instrukcji nielegalnych istnieją także instrukcje niezaimplementowane. Zaczynają się one od kodu %1010 (tzw. linia A-\$axxx) lub %1111 (linia F-\$fxxx). W wypadku natrafienia na jedną z nich, procesor wywołuje odpowiedni stan wyjątkowy. Każda z tych instrukcji ma przypisany odpowiedni wektor (A-10 i F-11). Do czego można to wykorzystać? Otóż dzięki temu, w prosty sposób mamy możliwość powiększenia zbioru instrukcji. W tym celu projektujemy format instrukcji i piszemy odpowiednią procedurę obsługi stanu wyjątkowego. Należy także uwzględnić to, że na stosie znajduje się licznik programu, który *w tym wypadku wskazuje dalej tą samą instrukcję*. Należy więc go odpowiednio zmodyfikować.

```

;-----
Start:
    MOVE.L    $10,old_vec
    MOVE.L    #Illegal_Instruction,$10          ;zapisz adres procedury
                                                ;obsługującej wektor nr 4
    DC.W      $bfff                            ;wywołaj przerwanie
                                                ;(nielegalna, nieistniejąca
                                                ;instrukcja)
    MOVE.L    Old_Vec,$10.w                    ;zapisz stary adres procedury
    RTS                                              ;wróć z programu

Illegal_Instruction:
    MOVEM.L   D0-A6,-(A7)                      ;rejstry na stos
    MOVE.W    #$fff,d7
Lp2:         MOVE.W    D7,$dff180                ;wygeneruj "kolorowy błysk"
    DBF       D7,Lp2
    MOVEM.L   (A7)+,D0-A6                      ;rejstry ze stosu

    ADD.L     #2,(A7)                          ;licznik programu na stosie+2
                                                ;ponieważ nielegalna instrukcja
                                                ;zajmuje dwa bajty
    RTE                                              ;powróć z przerwania

Old_Vec:DC.L    0
;-----

```



```

TRAP      #0                                ;wywołaj przerwanie
MOVE.L    Old_Vec,$80.w                    ;zapisz stary adres procedury
RTS                                              ;wróć z programu

Trap0_Instruction:
MOVE.M.L  D0-A6,-(A7)                      ;rejestry na stos
MOVE.W    #$fff,d7
Lp2:      MOVE.W    D7,$dff180              ;wygeneruj "kolorowy błysk"
          DBF       D7,Lp2
          MOVE.M.L  (A7)+,D0-A6             ;rejestry ze stosu
          RTE                                              ;powrót z przerwania

Old_Vec:DC.L    0

```

Rozkaz TRAP (Trap — pułapka)

Składnia:

TRAP #<wektor>

Atrybuty:

brak

Działanie:

Instrukcja TRAP odkłada na stos nadzorcy licznik programu PC, oraz rejestr statusowy SR. Procesor przełączany jest w stan nadzorcy, a do rejestru PC ładowany jest jeden z 16 wektorów TRAP, określony 4-bitowym polem instrukcji TRAP.

Kody warunków:

X — nie zmieniany,
 N — nie zmieniany,
 Z — nie zmieniany,
 V — nie zmieniany,
 C — nie zmieniany.

Dozwolone tryby adresowania:

żaden

9.5.2.5 Instrukcja TRAPV

Użycie instrukcji TRAPV spowoduje sprawdzenie przez mikroporcesor bitu V w rejestrze statusowym. Jeśli będzie on włączony, to wygenerowany zostanie stan wyjątkowy. Rozkaz ten jest stosowany w wypadku sprawdzenia prawdziwości otrzymanego wyniku.

Rozkaz TRAPV (Trap on overflow — pułapka w przypadku nadmiaru)

Składnia:

TRAPV

Atrybuty:

brak

Działanie:

Instrukcja TRAPV testuje bit nadmiaru i jeśli jest on ustawiony, to generuje stan wyjątkowy określony długim słowem spod adresu \$1c.

Kody warunków:

- X — nie zmieniany,
- N — nie zmieniany,
- Z — nie zmieniany,
- V — nie zmieniany,
- C — nie zmieniany.

Dozwolone tryby adresowania:

żaden

9.5.2.6 Instrukcja CHK

Następną instrukcją mogącą wywołać stan wyjątkowy, jest **CHK**. Sprawdza ona, czy operand pod adresem efektywnym posiada wartość mieszczącą się w danym zakresie. Jeśli jednak tak nie będzie, to wywołany zostanie stan wyjątkowy, którego nr wektora wynosi 6.

Rozkaz CHK (Check register against bounds — kontrola zawartości rejestru)**Składnia:**

CHK <ea>,Dn

Atrybuty:

rozmiar: W

Działanie:

Instrukcja CHK sprawdza, czy określony rejestr danych zawiera liczbę mieszczącą się w zadanym, dodatnim zakresie wartości. W przeciwnym razie generowany jest stan wyjątkowy.

Kody warunków:

- X — nie zmieniany,
- N — ustawiony, gdy zawartość rejestru danych jest mniejsza niż zero, zerowany, gdy zawartość rejestru danych jest większa niż operand określony adresem efektywnym, w przeciwnym razie niezdefiniowany,
- Z — nie zmieniany,
- V — nie zmieniany,
- C — nie zmieniany.

Dozwolone tryby adresowania:

Dn	An	(An)	(An)+	-(An)	x(An)	x(An.R.s)
tak	nie	tak	tak	tak	tak	tak
X.w	X.l	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	tak	tak	tak	nie	nie

9.5.2.7 Dzielenie przez zero

Dość specyficznym przypadkiem stanu wyjątkowego jest dzielenie przez 0. Wywołane jest ono, gdy użyjemy rozkazu DIVS #0,<ea> lub DIVU #0,<ea>. W wypadku gdy spróbujemy podzielić przez 0, wywołany zostanie stan wyjątkowy o wektorze nr 5.

9.5.3 Przerwania

Jak na wspominałem na początku, przerwania są stanami wyjątkowymi wywoływanymi zewnątrz, to znaczy nie przez mikroprocesor, ale copper czy blitter...

Przerwania możemy podzielić na dwie grupy:

- przerwania niemaskowalne, czyli takie, których zgłoszenie jest wykonywane *zawsze* i w każdych warunkach. Przerwanie to wykorzystuje opcja frezze w cartridgeach typu ACTION REPLAY.
- przerwania maskowalne, których wykonania możemy zabronić mikroprocesorowi, za pomocą ustawienia odpowiednich bitów, zwanych maską przerwań.

Może się zdarzyć, że równocześnie zostanie wywołane kilka przerwań. Aby ten problem rozwiązać, wprowadzono priorytety. W wyniku tego, gdy zostaną wywołane dwa przerwania, to najpierw zostanie wykonane przerwanie o wyższym priorytecie, a dopiero po jego wykonaniu, o niższym. Także w wypadku wykonywania przerwania, jego obsługa może zostać przerwana przez wystąpienie przerwania o wyższym priorytecie.

Użytkownik może zdecydować, czy mikroprocesor ma obsługiwać wszystkie przerwania, czy tylko te o priorytecie wyższym niż zadany. Służy do tego *maska przerwań* w rejestrze statusowym mikroprocesora. Są to trzy bity (od 8 do 10) — mamy więc do dyspozycji kombinacje od %000 do %111. Kombinacja %111 (czyli #7) określa, że mają być przyjmowane tylko przerwania niemaskowalne (NMI).

Priorytet przerwania określa jego poziom. Im wyższy poziom, tym większy priorytet mają przerwania na tym poziomie. Można sobie to wyobrazić na przykładzie siedmiopiętrowego wieżowca. Każde z jego pięter jest poziomem przerwań. Jednak na każdym piętrze może być kilka mieszkań (czyli kilka typów przerwań).

Jeśli zajrzysz teraz do tabeli stanów wyjątkowych, zauważysz, że poszczególnym poziomom przerwań przyporządkowano wektory od 25 do 31.

Oto jakie typy przerwań przyporządkowano poszczególnym poziomom:

Poziom	Numer wektora	Adres wektora	Nazwa przerwania	Bit w INTENA
1	25	\$64	SOFTIND	2
			DSKBLK	1
			TBE	0
2	26	\$68	PORTS	3
3	27	\$6c	COPER	4
			VERTB	5
			BLIT	6
4	28	\$70	AUD3	10
			AUD2	9
			AUD1	8
			AUD0	7
5	29	\$74	RBF	11
			DSKSYNC	12
6	30	\$78	EXTERN	13
			INTEN	14
7	31	\$7c	NMI	

Jeśli jednak dla jednego poziomu mamy kilka źródeł przerwań, to skąd mamy dokładnie wiedzieć, co było źródłem przerwania? Do tego celu zostały stworzone cztery rejestry INTENAR (\$dff01c), INTENA (\$dff09a) oraz INTREQR (\$dff01e), INTREQ (\$dff09c). Oto ich znaczenie.

Rejestry INTENA (W) i INTENAR (R) służą do decydowania, które przerwania mają być wykonywane, a które nie. Rejestr INTENAR służy do odczytu, a INTENA do zapisu.

bit 15 (SET/CLR) — Oznacza, czy wybrane bity mają być ustawione, czy skasowane. Jeśli do rejestru (dotyczy to tylko INTENA) zapiszemy jakąś wartość, to ten bit będzie oznaczał czy pozostałe bity mają być ustawione, czy skasowane. Zapiszmy na przykład taką liczbę:

```

%1001000110001000
|
SET/CLR

```

W tym wypadku wszystkie bity ustawione na jeden, będą w rejestrze zapalone, ponieważ wartość bitu SET/CLR wynosi 1. Pozostałe bity rejestru nie zostaną zmienione, a bit SET/CLR nie przyjmie żadnej wartości. Jednak gdybyśmy zapisali wartość: %0001000110001000, gdzie bit 15 jest skasowany, to wszystkie bity ustawione na jeden, zostałyby w tym rejestrze skasowane. Przypuśćmy, że mamy w rejestrze takie ustawienie bitów:

%0111111111111111

Po zapisaniu wartości %0001000110001000 do tego rejestru, otrzymamy:

zawartość rejestru	%0111111111111111
wpisywana wartość	%0001000110001000
nowa zawartość rejestru	%0110111001110111

Operacja ta jest bardzo pomocna, ponieważ na rejestrach układów specjalizowanych nie można operować na bitach.

- bit 14 (INTEN) — Umożliwia on blokadę wszystkich przerwań (0), za jednym zamachem. Nie jest to jednak wyłączenie przerwań. Ustawienie na 1 powoduje odblokowanie przerwań.
- bit 13 (EXTER) — Przerwanie generowane przez układ CIA-B i port rozszerzenia (1). Poziom 6.
- bit 12 (DSKSYN) — Znalezione wzorce synchronizacji dysku (1). Poziom 5.
- bit 11 (RBF) — Pełny bufor odbioru transmisji szeregowej (1). Poziom 5.
- bit 10 (AUD3) — Kanał dźwiękowy 3 (1). Poziom 4.
- bit 9 (AUD2) — Kanał dźwiękowy 2 (1). Poziom 4.
- bit 8 (AUD1) — Kanał dźwiękowy 1 (1). Poziom 4.
- bit 7 (AUD0) — Kanał dźwiękowy 0 (1). Poziom 4.
- bit 6 (BLIT) — Blitter gotowy (1). Poziom 3.
- bit 5 (VERTB) — Przerwa wygaszania pionowego (1). Poziom 3.
- bit 4 (COPER) — Przerwanie coppera (1). Poziom 3.
- bit 3 (PORTS) — Przerwanie generowane przez CIA-A i port rozszerzenia (1). Poziom 2.
- bit 2 (SOFT) — Software (1), czyli przerwanie programowe. Poziom 1.
- bit 1 (DSKBLK) — Blok danych z/do dysku, przesłane (1). Poziom 1.
- bit 0 (TBE) — Bufor przesyłania transmisji szeregowej jest pusty (1). Poziom 1.

Rejestry INTREQ i INTREQR mają identyczny opis, ale poszczególne bity są wykorzystywane do zgłoszenia danego przerwania. INTREQ służy do zgłoszenia przerwania, a z INTREQR możemy odczytać aktualny stan przerwań. Tak więc, jeśli chcemy zgłosić „ręcznie” przerwanie coppera, to patrzemy na powyższy opis i widzimy, że za przerwanie coppera odpowiada bit 4. Zapisujemy więc do INTREQ odpowiednią wartość, pamiętając że 15 bit tej wartości musi wynosić 1 (SET/CLR). Po obsłudze przerwania musimy skasować odpowiedni bit w rejestrze INTREQ zgłaszający to przerwanie. Poinformuje to system, że procedura przerwania została już zakończona.

9.5.3.1 Omówienie i przykłady inicjacji przerw

W celu lepszego zrozumienia zasad programowania procedur obsługi przerw, przedstawię przykładowe procedury, łącznie z charakterystyką niektórych z przerw poziomu 3 i 4. Pozostałe, rzadziej używane, omówię nieco później.

Poziom 3

Pewnie zauważyłeś, że w demach cało — lub więcej — dyskowych gra cały czas muzyka, a stacja dysków coś doczytuje, na ekranie szaleje „wektorówka”, itd. Jest tutaj wykorzystywane przerwanie VERTB. Jest ono generowane na początku *każdego* okresu wygaszania pionowego, czyli co każdą ramkę. Tak więc nie musimy czekać z procesorem na jakąś pozycję promienia, tylko główną procedurę zamieścić w procedurze obsługi przerwania.

Weźmy na przykład procedurę animacji z rozdziału 4.9. Mamy tam na początku następujące rozkazy:

```
Start:      MOVE.L  #Copper_List,$dff080      ; inicjuj copperlistę

Main_Loop:  BSR      Wait_Raster                ; odczekaj na wygaszanie pion.
            BSR      Bar_Sinus                  ; skocz do procedury animacji
                                                    ; rastra
            BTST.B   #6,$bfe001
            BNE      Main_Loop
            RTS
```

Jeśli teraz skasujemy **BSR WAIT_RASTER**, a **BSR Bar_Sinus** przeniesiemy do procedury obsługi przerwania, to wszystko będzie działać prawie tak samo. Główna różnica to to, że do pętli **Main_Loop** możemy dołożyć jakieś dodatkowe procedury.

A oto przykład wykorzystania przerwania VERTB.

```
;-----
Start:      MOVEM.L D0-A6,-(A7)                  ;rejestry na stos

            MOVE.L   4,A6
            JSR      -132(A6)                    ;przejmujemy kontrolę nad Amigą
                                                    ;o czym informujemy system
                                                    ;operacyjny

            MOVE.W   $dff01c,d0                  ;przechowujemy zawartość rej.
            OR.W      #$8000,d0                  ;INTENAR
            MOVE.W   D0,-(A7)                    ;zapisujemy ją na stos

            MOVE.W   #$7fff,$dff09a              ;zatrzymujemy przerwania

            MOVE.L   $6c,old_vec
            MOVE.L   #VERTB,$6c                  ;zapisz adres procedury
                                                    ;obsługującej wektor nr 27

            MOVE.W   #$c020,$dff09a              ;zezwól na przerwanie poziomu 3

Wait_M:     BTST.B   #6,$bfe001                  ;czekamy na przyciśnięty lewy
            BNE      Wait_M                      ;przycisk myszki

            MOVE.W   #$7fff,$dff09a              ;zatrzymaj przerwania
            MOVE.L   Old_Vec,$6c                 ;zapisz stary adres procedury
            MOVE.W   (A7)+,$dff09a               ;zapisz starą zawartość INTENAR
```



```

MOVE.L 4,A6 ;oddajemy kontrolę nad Amigą
JSR -138(A6) ;systemowi operacyjnemu

MOVEM.L (A7)+,D0-A6 ;rejestry ze stosu
MOVEQ #0,D0

RTS ;wróć z programu

VERTB:
MOVEM.L D0-A6,-(A7) ;rejestry na stos
MOVE.W $dff01e,d0 ;czy przerwanie na poziomie 3
BTST #5,D0 ;to VERTB???
BEQ No_Vblank ;jeśli nie, to nie obsługuj go
MOVE.W #$fff,d7
Lp2: MOVE.W D7,$dff180 ;wygeneruj "kolorowy błysk"
DBF D7,Lp2
MOVE.W #$0020,$dff09c ;informujemy system, że
;przerwanie VERTB zakończone

No_Vblank:
MOVEM.L (A7)+,D0-A6 ;rejestry ze stosu
RTE ;powrót z przerwania

Old_Vec:DC.L 0
;-----

```

Przerwanie COPER umożliwia copperowi zgłoszenie przerwania, poprzez copperlistę. Mimo, że copper ma możliwość zmiany każdego bitu w INTREQ (czyli wywołania każdego przerwania), to zarezerwowano dla niego dodatkowe przerwanie.

Aby wywołać przerwanie coppera, to musimy w copperliście umieścić następującą instrukcję:

```
DC.L $009c8010
```

Jeśli przedtem odczekamy na jakąś pozycję promienia wizji rozkazem WAIT, to co każdą ramkę, gdy promień osiągnie podane współrzędne, wygenerowane zostanie przerwanie.

Poziom 4

Na poziomie czwartym, mają miejsce przerwania AUDx. Informują one, że blok audio został zakończony. W trybie automatycznym generowane są one po zgłoszeniu się ostatniego słowa w strumieniu danych dla dźwięku. W trybie „ręcznym” występuje, gdy dany kanał jest gotowy do przyjęcia następnego słowa danych.

9.6 Układy CIA

Amiga posiada dwa układy służące do komunikacji ze światem zewnętrznym. Są to dwa identyczne układy CIA 8520, które jednak są wykorzystywane do różnych celów. Dla rozróżnienia nazywane są CIA-A i CIA-B.

Ten rozdział podzielony jest na dwie części. Pierwsza z nich omawia poszczególne rejestry układów CIA, a celem drugiej jest praktyczne wykorzystanie zdobytych wiadomości z zakresu tych układów (i nie tylko).

9.6.1 CIA-A

Amiga komunikuje się z układem CIA-A za pomocą rejestrów od \$bfe001 do \$bfef01. W odróżnieniu od rejestrów układów specjalizowanych jego rejestry mają długość 8 bitów, czyli jednego bajtu. Różnica dotyczy także tego, że każdy z rejestrów jest typu R/W, czyli można odczytać ich zawartość, jak i zapisać.

Oto oficjalne adresy 16 rejestrów wykorzystywanych przez CIA-A:

- \$bfe001 — PRA — rejestr danych zewnętrznych dla portu danych A,
- \$bfe101 — PRB — rejestr danych zewnętrznych dla portu danych B,
- \$bfe201 — DDRA — rejestr kierunku danych portu danych A,
- \$bfe301 — DDRB — rejestr kierunku danych portu danych B,
- \$bfe401 — TALO — młodszy bajt zegara A,
- \$bfe501 — TAHI — starszy bajt zegara A,
- \$bfe601 — TBLO — młodszy bajt zegara B,
- \$bfe701 — TBHI — starszy bajt zegara B,
- \$bfe801 — TODLO — młodszy bajt licznika TOD,
- \$bfe901 — TODMID — środkowy bajt licznika TOD,
- \$bfea01 — TODHI — starszy bajt licznika TOD,
- \$bfeb01 — TODHR — nie używany,
- \$bfec01 — SDR — rejestr danych szeregowych,
- \$bfed01 — ICR — rejestr kontroli przerwań,
- \$bfef01 — CRA — rejestr kontrolny A,
- \$bfef01 — CRB — rejestr kontrolny B.

Rejestr PRA (\$bfe001)

- bit 7 (FIR1) — przycisk „fire” w porcie joysticka 2 (0 — przycisk naciśnięty),
- bit 6 (FIR0) — przycisk „fire” w porcie joysticka 1 (0 — przycisk naciśnięty),
- bit 5 (RDY) — określa, czy stacja dysków jest gotowa do przyjmowania poleceń (1 — stacja nie jest gotowa; 0 — stacja jest gotowa),
- bit 4 (TK0) — czy głowica stacji dysków znajduje się nad ścieżką zerową? (1 — nie; 0 — tak),
- bit 3 (WPRO) — czy dysk w stacji jest zabezpieczony przed zapisem? (1 — nie; 0 — tak),
- bit 2 (CHNG) — czy dysk jest w stacji? (1 — bez zmiany; 0 — dysku nie ma),
- bit 1 (LED) — filtr dźwiękowy (1 — wyłączony; 0 — włączony),
- bit 0 (OVL) — bit przepełnienia pamięci (zawsze ustawiony na 0).

Bity 7 – 6 służą do rozpoznania, czy został wciśnięty lewy przycisk myszy lub fire w joysticku podłączonego do jednego z portów. Normalnie są ustawione jako wejście, co umożliwia aktualny odczyt danych. Zmieniając jednak ich kierunek na wyjście (ustawiając bity 7 i 6 w DDRA) możemy, zmieniając wartości tych bitów, przysyłać dane cyfrowe do jakiegoś urządzenia.

Bity 5 – 2 służą do komunikacji z kontrolerem stacji dysków. Ich znaczenie zostanie jednak podane nieco później, t.j. przy obsłudze stacji dysków. Bit 1 uruchamia filtr dźwiękowy Amigi, który powoduje obcięcie większości częstotliwości powyżej 7 kHz. Jego efektem jest także przyciemnienie (w starszych modelach wyłączenie), diody POWER w Amidze.

Rejestr DDRA (\$bfe201)

- bit 7 — wybór bitu 7 w PRA jako wejście/wyjście (0/1). Normalnie ustawiony na 0 (wejście),
- bit 6 — wybór bitu 6 w PRA jako wejście/wyjście (0/1). Normalnie ustawiony na 0 (wejście),
- bit 5 — wybór bitu 5 w PRA jako wejście/wyjście. Musi być ustawiony na 0 (wejście),
- bit 4 — wybór bitu 4 w PRA jako wejście/wyjście. Musi być ustawiony na 0 (wejście),
- bit 3 — wybór bitu 3 w PRA jako wejście/wyjście. Musi być ustawiony na 0 (wejście),
- bit 2 — wybór bitu 2 w PRA jako wejście/wyjście. Musi być ustawiony na 0 (wejście),
- bit 1 — wybór bitu 1 w PRA jako wejście/wyjście. Musi być ustawiony na 1 (wyjście),
- bit 0 — wybór bitu 0 w PRA jako wejście/wyjście. Musi być ustawiony na 1 (wyjście).

Bity w rejestrze DDRA określają, czy dane z rejestru PRA mają być pobierane ze swego źródła, czy też ich zapis spowoduje wysłanie danego bitu do źródła. Na przykład jeśli bity 7 i 6 ustawimy na 1, to ustawienie bitów 7 – 6 w PRA spowoduje pojawienie się sygnału fire w obydwu portach joysticka. Umożliwia to cyfrowe przesyłanie danych, itd..

Rejestr PRB (\$bfe101)

- bit 7 (D7) — złącze równoległe (PARALLEL PORT), pin 9.
- bit 6 (D6) — złącze równoległe (PARALLEL PORT), pin 8,
- bit 5 (D5) — złącze równoległe (PARALLEL PORT), pin 7,
- bit 4 (D4) — złącze równoległe (PARALLEL PORT), pin 6,
- bit 3 (D3) — złącze równoległe (PARALLEL PORT), pin 5,
- bit 2 (D2) — złącze równoległe (PARALLEL PORT), pin 4,
- bit 1 (D1) — złącze równoległe (PARALLEL PORT), pin 3,
- bit 0 (D0) — złącze równoległe (PARALLEL PORT), pin 2,

Rejestr DDRB (\$bfe301)

- bit 7 — wybór bitu 7 w PRB jako wejście/wyjście (0/1),
- bit 6 — wybór bitu 6 w PRB jako wejście/wyjście (0/1),
- bit 5 — wybór bitu 5 w PRB jako wejście/wyjście (0/1),
- bit 4 — wybór bitu 4 w PRB jako wejście/wyjście (0/1),
- bit 3 — wybór bitu 3 w PRB jako wejście/wyjście (0/1),
- bit 2 — wybór bitu 2 w PRB jako wejście/wyjście (0/1),
- bit 1 — wybór bitu 1 w PRB jako wejście/wyjście (0/1),
- bit 0 — wybór bitu 0 w PRB jako wejście/wyjście (0/1).

Rejestr PRB jest odpowiedzialny za kontrolę przesyłania danych przez port równoległy. Za pomocą odpowiedniego ustawienia bitów w rejestrze kontrolnym DDRB, możemy wysłać lub pobierać dane z urządzenia podłączonego do złącza. Np. jeśli chcemy wysłać bajt, to ustawiamy kierunek danych na wyjście (DDRB=\$ff) i zapisujemy odpowiednią wartość do PRB. Jeśli chcemy pobrać wartość z urządzenia, to zmieniamy kierunek na wejście i co jakiś okres czasu odcytujemy (próbkujemy) rejestr PRB.

Rejestry TALO (\$bfe401), TAHI (\$bfe501), TBLO (\$bfe601), TBHI (\$bfe701)

Układ CIA posiada dwa dwubajtowe zegary nazywane A i B, odliczające od podanej wartości do 0. Gdy je odczytujesz, to otrzymujesz aktualną wartość licznika zegara. Natomiast przy ich zapisie, ustalasz ich wartości początkowe, które zapisywane do tzw. przerzutników. Natomiast z przerzutników mogą być one zapisane do rejestru, po ustawieniu bitu LOAD w odpowiednim rejestrze sterującym. Każdy z zegarów składa się z dwóch bajtów: TxLO i TxHI, gdzie pierwszy z nich jest młodszym, a drugi starszym bajtem wartości zegara. Każdy z zegarów posiada własny rejestr sterujący. Dla zegara A jest to CRA, a dla B — CRB.

Z jaką szybkością zegary odliczają do zera? Otóż normalnie licznik jest zmniejszany o jeden co 10 cykli (taktów) mikroprocesora. Amidze z procesorem 68000, zabiera to około jedną milionową sekundy.

Gdy zegar A lub B odliczy do zera, powoduje to wygenerowanie przerwania, zasygnalizowanego w rejestrze \$bfed01. Jeśli przerwanie zegara jest udostępnione, to odpowiedni bit zostanie ustawiony w rejestrze INTREQ.

Jednak zegar A może odliczać także impulsy zewnętrzne na linii CNT, co omówimy jednak przy obsłudze klawiatury.

Zegar B może jednak nieco więcej. Może on liczyć, ile razy zegar A osiągnął 0, a także w sytuacji kiedy pojawił się sygnał na linii CNT i zegar A osiągnął 0.

Rejestry TODLO (\$bfe801 TODLO), TODMID (\$bfe901), TODHI (\$bfea01)

Układ CIA oprócz dwóch zegarów posiada także trzeci (zwany TOD), odliczający od zera w górę do \$ffffff (16777215). Gdy osiągnie on wartość maksymalną, rozpoczyna odliczanie od 0. Może być on wykorzystywany od wywołania przerwania, po osiągnięciu ustalonej wartości. Zegar ten powiększa wartość licznika co każdą ramkę. W trybie NTSC będzie to 60 razy na sekundę, a w PAL – 50.

Wartość licznika jest przechowywana trzech ośmiobitowych rejestrach:

- \$bfe801 — bity 0 – 7 licznika TOD,
- \$bfe901 — bity 8 – 15 licznika TOD,
- \$bfea01 — bity 16 – 23 licznika TOD.

Rejestry mogą być używane w dwóch celach. Przy odczycie zawsze otrzymamy aktualną wartość licznika. Natomiast przy zapisie rejestry te albo ustawiają licznik, albo określają przy jakiej wartości ma być wywołany *alarm*. Decydujemy o tym, ustawiając bit 7 w CRB.

Jednak należy się tutaj małe wyjaśnienie. Aby zapobiec zmianie licznika w chwili, gdy jest on odczytywany, zastosowano pewien ciekawy mechanizm. Otóż gdy odcytujemy najstarszy bajt, odświeżanie (ale tylko odświeżanie, bo liczenie trwa dalej) licznika zostaje zatrzymane do czasu,

aż zostanie odczytany najmłodszy bajt. W ten sposób spokojnie odczytujemy po kolei zawartość wszystkich rejestrów w takiej kolejności: TODHI, TODMID, TODLO — dzięki temu możemy być pewni prawidłowości wyniku.

Rejestr CRA (\$bfee01)

- bit 7 — nie używany,
- bit 6 (SPMODE) — tryb portu szeregowego (rejestr \$bfec01) (1 — wyjście, 0 — wejście)
- bit 5 (INMODE) — tryb zejścia zegara A (1 — liczenie sygnałów na linii CNT, 0 — odliczanie co 10 cykli mikroprocesora),
- bit 4 (LOAD) — załadowanie licznika zegara A wartością z przerzutnika A,
- bit 3 (RUNMODE) — tryb pracy zegara A (1 — jednorazowy, 0 — ciągły)
- bit 2 (OUTMODE) — tryb wyjścia portu danych B (1 — przełączenie bitu 6, 0 — impuls na bit 6 przez 10 cykli procesora),
- bit 1 (PBON) — wybór wyjścia zegara na port danych B (1 — wyjście zegara A pojawia się na bicie 6 dla portu danych B (\$bfe101)),
- bit 0 (START) — start zegara A (1 — start, 0 — stop)

Przy pomocy bitu 0 uruchamiamy lub zatrzymujemy zegar A. Aby uruchomić zegar czyścimy bit 0 (zatrzymujemy zegar), inicjujemy zegar, ustawiając jego wszystkie parametry (jak ma odliczać, zawartość przerzutnika, itp.). Dopiero po ustaleniu tych parametrów, zapisujemy 1 do bitu 0, czym uruchamiamy zegar.

Zegar A po odliczeniu do zera może zgłosić ten fakt, zmieniając bit 6 w rejestrze PRB (\$bfe101). Umożliwiamy to ustawiając bit PBON na 1, przy czym musimy określić, jak ma wyglądać informacja. Dzięki bitowi OUTMODE możemy określić jak ma wyglądać wyjście na rejestr PRB. Gdy ustawimy go na 0, to po dojściu licznika do 0, bit 6 w PRB będzie ustawiony na 1 przez 10 cykli mikroprocesora. Natomiast, gdy bit ten będzie wynosił 0, to bit 6 będzie przełączany, czyli jego wartość będzie się zmieniać z 1 na 0 i na odwrót. Dzięki temu, możemy wysyłać co jakiś okres czasu informację dla jakiegoś urządzenia zewnętrznego.

Można także określić tryb pracy zegara. Tryb ciągły występuje wtedy, gdy po odliczeniu do zera, licznik zegara zostaje załadowany wartością z przerzutnika i odliczanie rozpoczyna się od nowa. Natomiast tryb jednorazowy polega na tym, że po odliczeniu następuje zatrzymanie zegara.

Czasami potrzebujemy, aby natychmiast licznik zegara został załadowany wartością znajdującą się w przerzutniku. Możemy to zrobić, ustawiając bit LOAD.

Tryb zejścia zegara (bit 5) określa, kiedy wartość licznika zegara ma być zmniejszana o 1.

Bit 6 (SPMODE) zostanie omówiony, przy opisie klawiatury.

Rejestr CRB (\$bfef01)

- bit 7 (ALARM) — wybór stanu zapisu TOD (1—zapisywanie do rejestrów TOD ustawia licznik; 0—zapisywanie do rejestrów TOD ustawia alarm),
- bity 6 – 5 (INMODE) — tryb zejścia zegara B:
 - %00 — odliczanie co każde 10 cykli mikroprocesora,
 - %01 — liczenie sygnałów na linii CNT,

	%10	— odliczanie za każdym razem, jak zegar A osiąga zero,
	%11	— odliczanie gdy A osiąga zero i pojawia się sygnał na linii CNT,
bit 4 (LOAD)	—	załadowanie licznika zegara B wartością z przerzutnika B,
bit 3 (RUNMODE)	—	tryb pracy zegara B (1—jednorazowy; 0—ciągły)
bit 2 (OUTMODE)	—	tryb wyjścia portu danych B (1—przełączenie bitu 7; 0—impuls na bit 7 przez 10 cykli procesora),
bit 1 (PBON)	—	wybór wyjścia zegara na port danych B (1—wyjście zegara B pojawia się na bicie76 dla portu danych B (\$bfe101)),
bit 0 (START)	—	start zegara B (1—start; 0—stop)

Bity 0 – 4 w rejestrze CRB służą do tego samego celu, co odpowiednie bity w rejestrze CRA. Dotyczą jednak zegara B i bitu 7 w PRB (\$bfe101). Bit 7 kontroluje działanie rejestrów TOD. Ustawiając go na 1, ustawiamy czas alarmu zapisując rejestry TOD. Jeśli ustawimy go na 0, to zapisując rejestry TOD, ustawiamy ich zawartość.

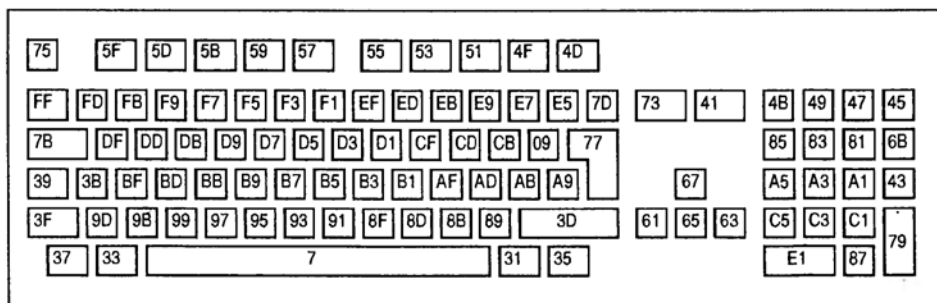
9.6.2 Przerwania układu CIA-A

Jeśli uważnie przeczytałeś rozdział o przerwaniach, pewnie zauważyłeś, że układ CIA-A ma możliwość wywołania przerwania. W związku z tym, że istnieje kilka źródeł powodujących przerwanie, stworzono dodatkowy zarządzający nimi rejestr — ICR (\$bfd01).

Aby rozpoznać konkretne źródło przerwania, musimy w procedurze obsługi przerwania PORTS odczytać zawartość rejestru ICR i sprawdzać kolejne źródła. Należy jednak pamiętać, by po odczycie rejestru ten ulega skasowaniu. Nie musimy więc już dbać o informowanie zakończenia obsługi przerwania (ale bit je zgłaszający w INTREQ, musimy skasować).

A oto znaczenie poszczególnych bitów w rejestrze ICR.

- Przy odczycie:
 - bit 0 — czy zegar A odliczył do 0? (1 — tak),
 - bit 1 — czy zegar B odliczył do 0? (1 — tak),
 - bit 2 — czy zegar TOD osiągnął czas alarmu? (1 — tak),
 - bit 3 — czy rejestr SDR skończył pobieranie/wysyłanie danych? (1 — tak),
 - bit 4 — czy został wysłany sygnał na linię FLAG (port rozszerzenia)? (1 — tak),
 - bit 5 — nie używany,
 - bit 6 — nie używany,
 - bit 7 — czy jakieś źródło z CIA-A powoduje przerwanie? (1 — tak)
- Przy zapisie:
 - bit 0 — przerwanie zegara A: 1 — udostępnione, 0 — nieudostępnione,
 - bit 1 — przerwanie zegara B: 1 — udostępnione, 0 — nieudostępnione,
 - bit 2 — przerwanie zegara TOD (czas alarmu): 1 — udostępnione, 0 — nieudostępnione,
 - bit 3 — przerwanie SDR: 1 — udostępnione, 0 — nieudostępnione,
 - bit 4 — przerwanie linii FLAG: 1 — udostępnione, 0 — nieudostępnione,
 - bit 5 — nie używany,
 - bit 6 — nie używany,
 - bit 7 — bit SET/CLR — wszystkie bity będą ustawione zgodnie z tą wartością.



Oprócz zwykłych kodów, istnieją jeszcze kody specjalne, których zadaniem jest przesłanie komunikatów o błędach:

- \$0f — została wykryta kombinacja CONTROL-AMIGA-AMIGA,
- \$0c — ostatni kod klawisza jest zniekształcony (synchronizacja została utracona, po czym ponownie odzyskana),
- \$0a — 10-cio znakowy bufor klawiatury jest już wypełniony (należy go opróżnić),
- \$06 — klawiatura jest uszkodzona (autotest klawiatury wykazał błąd),
- \$04,\$02 — jeśli podczas włączania komputera, zostały naciśnięte jakieś klawisze, to najpierw zostaje przesłany kod \$04, następnie kody wszystkich naciśniętych klawiszy, a po tym kod \$02.

Oto przykładowa procedura odczytująca kody klawiszy z rejestru SDR.

```

;-----
Start:  MOVEM.L D0-A6, - (A7)                ;rejestry na stos

        MOVE.L 4,A6
        JSR    -132(A6)                    ;przejmujemy kontrolę nad Amigą
                                           ;o czym informujemy system
                                           ;operacyjny

        MOVE.L 156(A6),A1
        MOVE.L 38(A1),Old_Copper           ;przechowaj adres copperlisty
                                           ;systemu operacyjnego

        MOVE.L #Copper,$dff080            ;inicjujemy copperlistę

        MOVE.W $dff01c,d0                  ;przechowujemy zawartość rej.
        OR.W   #$8000,d0                   ;INTENAR
        MOVE.W D0,-(A7)                   ;zapisujemy ją na stos

        MOVE.B #%01111111,$bfed01         ;kasujemy wszystkie źródła
        MOVE.B #%10001000,$bfed01         ;przerwań CIA-A
                                           ;umożliwiamy tylko przerwanie SDR

        MOVE.W #$7fff,$dff09a             ;zatrzymujemy przerwania
        MOVE.L $68,-(a7)                   ;wektor 26 na stos

        MOVE.L #PORTS,$68                  ;nowa procedura obsługi wek. 26
        MOVE.W #$c008,$dff09a             ;zezwól na przerwanie poziomu 2

Main_Loop:
        BTST   #6,$bfe001                  ;czy lewy przycisk myszki?
        BNE.B  Main_Loop                   ;jeśli nie, to do skocz pętli

End:    MOVE.W #$7fff,$dff09a              ;zatrzymaj przerwania
        MOVE.L (A7)+,$68                   ;ze stosu pobierz stary wek. 26
        MOVE.W (A7)+,$dff09a              ;zapisz starą zawartość INTENAR
        MOVE.B #%11111111,$bfed01         ;zezwól na wszystkie przerwania
                                           ;CIA-A
        MOVE.L Old_Copper,$dff080         ;uruchamiamy starą copperlistę

        MOVE.L 4,A6                        ;oddajemy kontrolę nad Amigą
        JSR    -138(A6)                    ;systemowi operacyjnemu

        MOVEM.L (A7)+,D0-A6                ;rejestry ze stosu
        MOVEQ   #0,D0                      ;powrót z programu
        RTS

```



```

PORTS:
    MOVEM.L D0-A6, -(A7)                ;rejstry na stos
    MOVE.W $dff01e,d0                  ;czy przerwanie na poziomie 2
    BTST #3,D0                          ;to CIA-A???
    BEQ Int_Exit                        ;jeśli nie, to nie obsługuj go

    MOVE.B $bfed01,d0                  ;to przerwanie SDR???
    BTST #3,D0                          ;jeśli nie, to nie obsługuj go
    BEQ Int_Exit

    MOVE.B $bfec01,d0                  ;pobierz daną z SDR
    ROR.B #1,D0
    EOR.B #$ff,d0                      ;w D0 RAWKEY
    MOVE.W D0,Copper+2                 ;potraktuj go jako kolor i
                                        ;zapisz jako drugie słowo
                                        ;instrukcji MOVE w copperliście
    OR.B 0x01000000,$bfee01            ;ustawiamy SDR na wyjście
    MOVE.B 0x0,$bfec01                ;wysyłamy Handshake
    EOR.B 0x01000000,$bfee01           ;ustawiamy SDR na wejście
    MOVE.W 0x8,$dff09c                 ;informujemy system, że
                                        ;przerwanie PORTS zakończone

Int_Exit:
    MOVEM.L (A7)+,D0-A6                ;rejstry ze stosu
    RTE                                ;powrót z przerwania

Copper: DC.L 0x01800000                 ;zmiana koloru tła. Procedura
                                        ;obsługi przerwania wpisuje do
                                        ;drugiego słowa instrukcji jako
                                        ;kolor - kod klawisza
    DC.L 0x01000000                    ;pustka na ekranie
    DC.L 0xffffffff                    ;koniec copperlisty
Old_Copper: DC.L 0
;-----

```

Znaczną część tego przykładu, zajęło tzw. „zabicie systemu operacyjnego”. Cóż to oznacza? Otóż system operacyjny Amigi (czyli Amiga DOS, Workbench), wykorzystuje do swoich celów układy CIA, przerwania, itd. Aby po wyjściu z naszego programu wszystko działało prawidłowo, musimy odtworzyć stan pierwotny. Tak więc zapisujemy dokładnie stan przerwań, copperlistę i informujemy system o przejęciu kontroli nad komputerem. Dokładne objaśnienia tych zagadnień znajdują się w rozdziale „Współpraca z systemem operacyjnym”.

Rozkaz EOR (Exclusive OR — różnica symetryczna)

Składnia:

EOR Dn,<ea>

Atrybuty:

rozmiar: B, W, L

Działanie:

Instrukcja EOR przeprowadza działanie różnicy symetrycznej (exclusive or) pomiędzy rejestrem danych, a operandem przeznaczenia określonym adresem efektywnym. Wynik operacji zapisany jest w miejscu przeznaczenia.

Na czym polega różnica symetryczna? Oto przykład:

$$\begin{array}{r}
 \%11111111 \\
 \text{EOR } \%01010000 \\
 \hline
 \%10101111
 \end{array}$$

Można z niego wywnioskować, że:

$$0 \text{ EOR } 0 = 0$$

$$0 \text{ EOR } 1 = 1$$

$$1 \text{ EOR } 0 = 1$$

$$1 \text{ EOR } 1 = 0$$

Można zauważyć, że jeśli dokonamy operacji EOR z jakąś liczbą i z wynikiem postąpimy tak samo, to wynikiem będzie pierwsza liczba. Oto przykład:

$$\begin{array}{r} \%00011101 \\ \text{EOR } \%10101101 \\ \hline \%10110000 \\ \text{EOR } \%10101101 \\ \hline \%00011101 \end{array}$$

Ta zależność jest bardzo często wykorzystywana przy kodowaniu danych.

Kody warunków:

X — nie zmieniany,

N — ustawiany, gdy najstarszy bit wyniku wynosi 1, w przeciwnym wypadku kasowany,

Z — ustawiany, gdy wynik wynosi 0, w przeciwnym wypadku kasowany,

V — zawsze zerowany,

C — zawsze zerowany.

Dozwolone tryby adresowania:

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
tak	nie	tak	tak	tak	tak	tak
X.w	X.1	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	nie	nie	nie	nie	nie

Przykład:

```

LEA     Dane_Źródłowe,A0
MOVE.W #2048-1,D7
MOVE.W #$abcd,D0
Loop:  EOR.W D0,(A1)+
        DBF D7,Loop
        RTS

```

Powyższy przykład spowoduje zakodowanie 4 kilobajtów danych leżących pod adresem określonym przez etykietę Dane_Źródłowe. Powtórne uruchomienie tej procedury spowoduje odzyskanie zakodowanych danych.

Rozkaz EORI (Exclusive OR Immediate — natychmiastowa różnica symetryczna)**Składnia:**

EORI #<dana>,<ea>

Atrybuty:

rozmiar: B, W, L

Działanie:

Instrukcja EOR przeprowadza działanie różnicy symetrycznej (exclusive or) pomiędzy daną natychmiastową, a operandem przeznaczenia określonym adresem efektywnym. Wynik operacji zapisany jest w miejscu przeznaczenia.

Kody warunków:

X — nie zmieniany,

N — ustawiany, gdy najstarszy bit wyniku wynosi 1, w przeciwnym wypadku kasowany,

Z — ustawiany, gdy wynik wynosi 0, w przeciwnym wypadku kasowany,

V — zawsze zerowany,

C — zawsze zerowany.

Dozwolone tryby adresowania:

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
tak	nie	tak	tak	tak	tak	tak
X.w	X.l	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	nie	nie	nie	nie	nie

Przykład:

EORI.W #\$ffff,D0

Zawartość młodszego słowa rejestru D0 zostanie zanegowana.

9.6.4 CIA-B

Układy CIA-A i CIA-B, praktycznie nie różnią się od siebie. Jediną różnicą, jest ich wykorzystanie przez Amigę. W związku z tym, omówienie układu CIA-B będzie się ograniczało, tylko do omówienia różnic.

Amiga komunikuje się z układem CIA-B za pomocą rejestrów od \$bfd000 do \$bfd00.

Oto adresy 16 rejestrów wykorzystywanych przez CIA-B:

- \$bfd000 — PRA — rejestr danych zewnętrznych dla portu danych A,
- \$bfd100 — PRB — rejestr danych zewnętrznych dla portu danych B,
- \$bfd200 — DDRA — rejestr kierunku danych portu danych A,
- \$bfd300 — DDRB — rejestr kierunku danych portu danych B,
- \$bfd400 — TALO — młodszy bajt zegara A,
- \$bfd500 — TAHI — starszy bajt zegara A,
- \$bfd600 — TBLO — młodszy bajt zegara B,
- \$bfd700 — TBHI — starszy bajt zegara B,
- \$bfd800 — TODLO — młodszy bajt licznika TOD,
- \$bfd900 — TODMID — środkowy bajt licznika TOD,
- \$bfda00 — TODHI — starszy bajt licznika TOD,
- \$bfdb00 — TODHR — nie używany,
- \$bfdd00 — SDR — rejestr danych szeregowych,
- \$bfdd00 — ICR — rejestr kontroli przerwań,
- \$bfde00 — CRA — rejestr kontrolny A,
- \$bfdf00 — CRB — rejestr kontrolny B.

Rejestr PRA (\$bfd000)

- bit 7 (DTR) — terminal danych gotowy dla złącza RS-232 (pin 20 portu szeregowego),
- bit 6 (RTS) — zgłoszenie wysłania danych dla złącza RS-232 (pin 4 portu szeregowego),
- bit 5 (DCD) — wykrycie fali nośnej dla złącza RS-232 (pin 8 portu szeregowego),
- bit 4 (CTS) — wyczyszczenie w celu przestania danych dla złącza RS-232 (pin 5 portu szeregowego),
- bit 3 (DSR) — gotowość danych dla złącza RS-232 (pin 6 portu szeregowego),
- bit 2 (SEL) — wybór złącza CENTRONICS (pin 13 portu równoległego),
- bit 1 (POUT) — koniec papieru (pin 12 portu równoległego),
- bit 0 (BUSY) — przez złącze CENTRONICS są przesyłane dane (pin 11 portu równoległego).

Rejestr DDRA (\$bfd200)

- bit 7 — wybór bitu 7 w PRA jako wejście/wyjście (0/1). Normalnie ustawiony na 1 (wyjście),
- bit 6 — wybór bitu 6 w PRA jako wejście/wyjście (0/1). Normalnie ustawiony na 1 (wyjście),
- bit 5 — wybór bitu 5 w PRA jako wejście/wyjście (0/1). Normalnie ustawiony na 0 (wejście),
- bit 4 — wybór bitu 4 w PRA jako wejście/wyjście (0/1). Normalnie ustawiony na 0 (wejście),
- bit 3 — wybór bitu 3 w PRA jako wejście/wyjście (0/1). Normalnie ustawiony na 0 (wejście),
- bit 2 — wybór bitu 2 w PRA jako wejście/wyjście (0/1). Normalnie ustawiony na 0 (wejście),
- bit 1 — wybór bitu 1 w PRA jako wejście/wyjście (0/1). Normalnie ustawiony na 0 (wejście),
- bit 0 — wybór bitu 0 w PRA jako wejście/wyjście (0/1). Normalnie ustawiony na 0 (wejście),

Rejestr PRA jest używany do kontroli stanu linii portów: szeregowego i równoległego. Oczywiście istnieje możliwość użycia każdego z tych bitów jako wejścia lub wyjścia. Dokonujemy tego za pomocą rejestru DDRA.

Rejestr PRB (\$bfd100)

- bit 0 (STEP) — przesunięcie głowicy stacji o jedną ścieżkę w kierunku określonym przez bit DIR (należy ustawić: 1, 0, 1),
- bit 1 (DIR) — kierunek przesunięcia głowicy (1 — na zewnątrz, 0 — do środka),
- bit 2 (SIDE) — wybór głowicy stacji dysków (1 — dolna, 0 — górna),
- bit 3 (SEL0) — wybór stacji DF0: (1 — nie jest wybrana, 0 — wybrana),
- bit 4 (SEL1) — wybór stacji DF1: (1 — nie jest wybrana, 0 — wybrana),
- bit 5 (SEL2) — wybór stacji DF2: (1 — nie jest wybrana, 0 — wybrana),
- bit 6 (SEL3) — wybór stacji DF3: (1 — nie jest wybrana, 0 — wybrana),
- bit 7 (MTR) — stan silnika (1 — silnik wyłączony, 0 — silnik włączony).

Rejestr DDRB (\$bfd300)

- bit 7 — wybór bitu 7 w PRB jako wejście/wyjście (0/1). Musi być ustawiony na 1 (wyjście),
- bit 6 — wybór bitu 6 w PRB jako wejście/wyjście (0/1). Musi być ustawiony na 1 (wyjście),
- bit 5 — wybór bitu 5 w PRB jako wejście/wyjście (0/1). Musi być ustawiony na 1 (wyjście),
- bit 4 — wybór bitu 4 w PRB jako wejście/wyjście (0/1). Musi być ustawiony na 1 (wyjście),
- bit 3 — wybór bitu 3 w PRB jako wejście/wyjście (0/1). Musi być ustawiony na 1 (wyjście),
- bit 2 — wybór bitu 2 w PRB jako wejście/wyjście (0/1). Musi być ustawiony na 1 (wyjście),
- bit 1 — wybór bitu 1 w PRB jako wejście/wyjście (0/1). Musi być ustawiony na 1 (wyjście),
- bit 0 — wybór bitu 0 w PRB jako wejście/wyjście (0/1). Musi być ustawiony na 1 (wyjście).

Rejestr PRB jest wykorzystywany przez Amigę w celu komunikacji z kontrolerem stacji dysków. Dzięki niemu możemy wykonywać podstawowe operacje, takie jak przesuwanie głowicy, uruchamianie i zatrzymywanie silnika, wybór stacji.

Dokładne omówienie pracy stacji dysków, przykładowe procedury kontroli, znajdują się w rozdziale 9.6.5.

Rejestr TALO (\$bfd400), TAHI (\$bfd500), TBLO (\$bfd600), TBHI (\$bfd700)

Zegary A i B w układzie CIA-B są identyczne do analogicznych zegarów znajdujących się w CIA-A. Nie są one wykorzystywane przez Amigę, możesz więc je używać do własnych celów.

Rejestr TODLO (\$bfd800), TODMID (\$bfd900), TODHI (\$bfda00)

Zegar TOD w układzie CIA-B zwiększa zawartość licznika o jeden, po każdej przerwie wygaszania poziomego — około 31500 razy na sekundę. Poza tym nie różni się on od zegara TOD w układzie CIA-A.

Wartość licznika jest przechowywana trzech ośmiobitowych rejestrach:

- \$bfd800 — bity 0 – 7 licznika TOD,
- \$bfd900 — bity 8 – 15 licznika TOD,
- \$bfda00 — bity 16 – 23 licznika TOD.

Rejestr CRA (\$bfde00)

bit 7	— nie używany,
bit 6 (SPMODE)	— tryb portu szeregowego (rejestr \$bfdc00) (1 — wyjście, 0 — wejście)
bit 5 (INMODE)	— tryb zejścia zegara A (1 — liczenie sygnałów na linii CNT, 0 — odliczanie co 10 cykli mikroprocesora),
bit 4 (LOAD)	— załadowanie licznika zegara A wartością z przerzutnika A,
bit 3 (RUNMODE)	— tryb pracy zegara A (1 — jednorazowy, 0 — ciągły)
bit 2 (OUTMODE)	— tryb wyjścia portu danych B (1 — przełączenie bitu 6, 0 — impuls na bit 6 przez 10 cykli procesora),
bit 1 (PBON)	— wybór wyjścia zegara na port danych B (1 — wyjście zegara A pojawia się na bicie 6 dla portu danych B (\$bfd100)),
bit 0 (START)	— start zegara A (1 — start, 0 — stop)

Rejestr CRB (\$bfdf00)

bit 7 (ALARM)	— wybór stanu zapisu TOD (1 — zapisywanie do rejestrów TOD ustawia licznik, 0 — zapisywanie do rejestrów TOD ustawia alarm),
bity 6 – 5 (INMODE)	— tryb zejścia zegara B: %00 — odliczanie co każde 10 cykli mikroprocesora, %01 — liczenie sygnałów na linii CNT, %10 — odliczanie za każdym razem, jak zegar A osiąga zero, %11 — odliczanie gdy A osiąga zero i pojawia się sygnał na linii CNT,
bit 4 (LOAD)	— załadowanie licznika zegara B wartością z przerzutnika B,
bit 3 (RUNMODE)	— tryb pracy zegara B (1 — jednorazowy, 0 — ciągły)
bit 2 (OUTMODE)	— tryb wyjścia portu danych B (1 — przełączenie bitu 7, 0 — impuls na bit 7 przez 10 cykli procesora),
bit 1 (PBON)	— wybór wyjścia zegara na port danych B (1 — wyjście zegara B pojawia się na bicie 7 dla portu danych B (\$bfd100)),
bit 0 (START)	— start zegara B (1 — start, 0 — stop)

Znaczenie rejestrów CRA i CRB jest identyczne jak w wkładzie CIA-A.

Rejestr ICR (\$bfdd00)

- Przy odczycie:
 - bit 0 — czy zegar A odliczył do 0? (1 — tak),
 - bit 1 — czy zegar B odliczył do 0? (1 — tak),
 - bit 2 — czy zegar TOD osiągnął czas alarmu? (1 — tak),
 - bit 3 — czy rejestr SDR skończył pobieranie/wysyłanie danych? (1 — tak),
 - bit 4 — czy został wysłany sygnał na linię FLAG (port rozszerzenia)? (1 — tak),
 - bit 5 — nie używany,
 - bit 6 — nie używany,
 - bit 7 — czy jakieś źródło z CIA-B powoduje przerwanie? (1 — tak)

Przy zapisie:

- bit 0 — przerwanie zegara A: 1 — udostępnione, 0 — nieudostępnione,
- bit 1 — przerwanie zegara B: 1 — udostępnione, 0 — nieudostępnione,
- bit 2 — przerwanie zegara TOD (czas alarmu): 1 — udostępnione, 0 — nieudostępnione,
- bit 3 — przerwanie SDR: 1 — udostępnione, 0 — nieudostępnione,
- bit 4 — przerwanie linii FLAG: 1 — udostępnione, 0 — nieudostępnione,
- bit 5 — nie używany,
- bit 6 — nie używany,
- bit 7 — bit SET/CLR — wszystkie bity będą ustawione zgodnie z tą wartością.

Różnica pomiędzy CIA-A i CIA-B, dotycząca tego rejestru, jest taka, że układ CIA-B generuje przerwanie EXTER na poziomie 6. Poza tym, nie ma żadnych innych różnic.

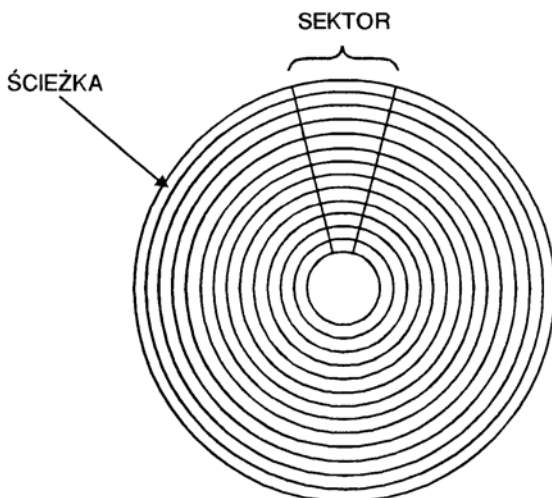
9.6.5 Kontrola stacji dysków

Z tego rozdziału dowiesz się, jak kontrolować stację dysków oraz odczytywać i zapisywać dane bez użycia systemu operacyjnego.

9.6.5.1 Pojemność dyskietki, a kod MFM

Dyskietki stosowane w Amidze są dwustronne. Każda ze stron ma po 80 ścieżek, a każda ścieżka jest podzielona na 11 sektorów. Sektor może przechować 512 bajtów danych zakodowanych w systemie MFM. Daje to w sumie pojemność:

$$\begin{aligned} & (2 \text{ strony}) \cdot (80 \text{ ścieżek}) \cdot (11 \text{ sektorów}) \cdot (512 \text{ bajtów}) = \\ & = (1760 \text{ sektorów}) \cdot (512 \text{ bajtów}) = 901120 \text{ bajtów} = 880 \text{ kB} \end{aligned}$$



Jeśli jednak chcemy kontrolować poprawność zapisu lub odczytu danych, to musimy wprowadzić system poprawności zapisu i odczytu. Do tego celu został utworzony specjalny kod zapisu danych na nośnikach magnetycznych. Jest to system MFM (ang. *Modified Frequency Modulation*).

Jego główną zasadą jest to, że w zapisie bitowym danej nie mogą wystąpić obok siebie dwie jedyńki:

dobrze %100010101001010 %10001000

źle %110001001010011 %11111100

Trzeba więc wszystkie dane zakodować w taki sposób, by nie występowały obok siebie dwie jedyńki. Robi się to w taki sposób, że gdy dwa sąsiednie bity mają wartość 0 to między nie wstawia się 1, w każdym innym przypadku wstawia się 0.

Dana przed kodowaniem %1 0 0 1 1 1 0 1 0 1 0 1 0 0 0

Dana po kodowaniu %10010010101000100010001001010

Aby nieco ułatwić sprawę, bity rozбивa się na parzyste i nieparzyste i koduje się je osobno:

Dana przed kodowaniem %1 0 0 1 1 1 0 1 0 1 0 1 0 0 0

Bity nieparzyste %1 0 1 0 0 0 0 0

Bity parzyste % 0 1 1 1 1 1 0

Zakodowane bity nieparzyste %1 0 0 0 1 0 0 1 0 1 0 1 0 1 0

Zakodowane bity parzyste % 0 0 1 0 1 0 1 0 1 0 1 0 0

Widać więc, że po zakodowaniu dana jest dwukrotnie dłuższa od danej oryginalnej. Wobec tego, okazuje się, że w rzeczywistości dyskietka ma pojemność dwukrotnie większą, czyli 1720 kB

Jednak po doczytaniu ścieżki do pamięci, mamy 11 sektorów, z których każdy ma długość 576 bajtów. Na pierwsze 64 bajty, składają się różne dodatkowe informacje (numer sektora, numer ścieżki, sumy kontrolne, itp.). Oto pełny opis wyglądu sektora.

Offset (Przesunięcie)	Zawartość	
\$000 (00)	Długie słowo	\$aaaaaaa
\$004 (04)	2 słowa synchronizacji	\$44894489
008 (08)	Format Byte	(bity nieparzyste)
\$009 (09)	Numer ścieżki	(bity nieparzyste)
\$00a (10)	Numer sektora	(bity nieparzyste)
\$00b (11)	Sektory	(bity nieparzyste)
\$00c (12)	Bajt formatu	(bity parzyste)
\$00d (13)	Numer ścieżki	(bity parzyste)

Offset (Przesunięcie)	Zawartość	
\$00e (14)	Numer sektora	(bity parzyste)
\$00f (15)	Sektory	(bity parzyste)
\$010 (16)	"Sector Label"	(bity nieparzyste)
\$020 (32)	"Sector Label"	(bity parzyste)
\$030 (48)	Nagłówek sumy kontrolnej	(bity nieparzyste)
\$034 (52)	Nagłówek sumy kontrolnej	(bity parzyste)
\$038 (56)	Suma kontrolna	(bity nieparzyste)
\$03c (60)	Suma kontrolna	(bity parzyste)
\$040 (64)	512 bajtów danych	(bity nieparzyste)
\$240 (576)	512 bajtów danych	(bity parzyste)
Długość sektora:	1088 (\$440) bajtów	

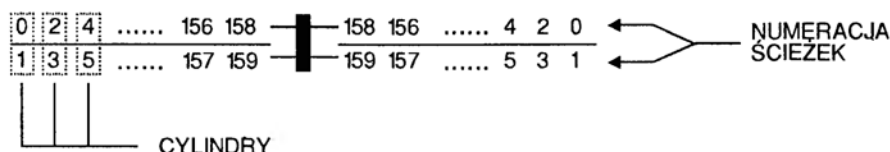
Tak więc, rzeczywista pojemność dyskietki wynosi 1870 kB.

9.6.5.2 Obsługa kontrolera stacji dysków

Aby cokolwiek nagrać lub odczytać z dyskietki, musimy najpierw ustawić głowicę w odpowiednim miejscu dysku. Należy więc wykonać następujące kroki:

- wybrać stację dysków (od Df0: do Df3:),
- sprawdzić obecność dyskietki w stacji,
- włączyć silnik,
- w celu orientacji znaleźć ścieżkę 0,
- przesunąć głowicę do żądanej ścieżki.

Amiga nie ma możliwości sprawdzenia, w jakim miejscu znajduje się aktualnie głowica stacji dysków — dlatego najpierw odnajdujemy ścieżkę zerową. Oto numeracja ścieżek na dyskietce.



Jak widać, im bliżej środka dysku, tym wyższy numer ścieżki.

Oprócz ścieżek, na dyskietce wyróżniamy także cylindry. Są to po prostu dwie odpowiadające sobie ścieżki: górna i dolna (na przykład ścieżki 0 i 1).

Do kontroli stacji dysków będą przydatne dwa rejestry: PRB układu CIA-B i PRA układu CIA-A.

Uwaga: aby obsługiwać stację dysków z poziomu CIA, musimy najpierw przejąć **całkowitą** kontrolę nad systemem operacyjnym. W tym wypadku wystarczy całkowite wyłączenie (lub blokada) przerwań.

A oto przykładowe procedury kontroli dysku, ze szczegółowym opisem.

a. Włączanie silnika i wybór stacji dysków

```
Df00n:  OR.B    #$fff,$bfd100      ;wszystkie stacje nieaktywne, silnik
                                     ;wyłączony
        BCLR   #7,$bfd100        ;włączenie silnika
        NOP
        NOP
        BCLR   #3,$bfd100        ;aktywna stacja Df0:
        RTS
```

Bity 3 – 6 w rejestrze PRB układu CIA-B, służą do wyboru stacji dysków. Pozostałe bity tego rejestru oddziałują tylko na wybrane tutaj stacje. Oznacza to, że możemy równolegle przesuwac głowice równocześnie w kilku stacjach, jednak nie ma to praktycznego wykorzystania.

Aby wybrać odpowiednią stację, musimy ustawić bity 3 – 6 i upewnić się, że silnik jest wyłączony. Potem włączamy silnik czyszcząc bit 3 rejestru PRB i wybieramy żadaną stację.

Rozkaz NOP (No Operation — bez działania)

Składnia:

NOP

Atrybuty:

brak

Działanie:

Instrukcja NOP nie zmienia nic oprócz licznika programu, który zostaje ustawiony na następną instrukcję. Instrukcje NOP są używane w celu wprowadzenia opóźnień w programie.

Kody warunków:

X — nie zmieniany,

N — nie zmieniany,

Z — nie zmieniany,

V — nie zmieniany,

C — nie zmieniany.

Dozwolone tryby adresowania:

żaden

b. Oczekiwanie na gotowość stacji dysków

Po włączeniu stacji dysków, musimy odczekać aż będzie ona gotowa do przyjmowania poleceń. W tym celu wystarczy tylko testować bit RDY w rejestrze PRA układu CIA-A.

```
DWait:  BTST    #5,$bfe001    ;testuj bit 5 rejestru PRA
        BNE.S   DWait        ;jeśli =1, to czekaj dalej
        RTS
```

c. Wybór głowic

Skoro dyski w Amidze są dwustronne, stacje dysków muszą mieć dwie głowice: dolną i górną. Zależnie od tego którą wybierzemy, ta głowica będzie odczytywać lub zapisywać dane.

Wyboru głowicy dokonujemy za pomocą bitu 2 w rejestrze \$bfd100. Jeśli bit ten ustawimy, to wybierzemy głowicę dolną. Jeśli go wyzerujemy, to wybierzemy głowicę górną.

W taki sposób można wybierać głowice:

```
Głowica_Górna:
    BCLR    #2,$bfd100
Głowica_Dolna:
    BSET    #2,$bfd100
```

Oto instrukcje manipulacji bitami, które wykorzystaliśmy:

Rozkaz BCHG (Test A Bit and Change — testowanie bitu i zamiana)

Składnia:

```
BCHG  #<dana>,<ea>
BCHG  Dn,<ea>
```

Atrybuty:

rozmiar: B, L

Działanie:

Instrukcja BCHG testuje określony bit operandu przeznaczenia i w zależności od jego wartości, ustawia bit Z rejestru statusowego. Potem dany bit operandu zostaje zanegowany. Numer bitu do testowania może być określony przez daną natychmiastową (testowanie statyczne) lub zawartość określonego rejestru danych (testowanie dynamiczne). W wypadku testowania operandu w pamięci, instrukcja ma rozmiar bajtu. Gdy będziemy testować bit rejestru danych, ma ona rozmiar długiego słowa.

Kody warunków:

X — nie zmieniany,
N — nie zmieniany,
Z — ustawiany, gdy testowany bit był zerowy,
V — nie zmieniany,
C — nie zmieniany.

Dozwolone tryby adresowania:

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
tak	nie	tak	tak	tak	tak	tak
X.w	X.1	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	nie	nie	nie	nie	nie

Przykład:

BCHG #5, D0

5-ty bit rejestru D0 zostanie przetestowany, a następnie zanegowany.

Rozkaz BCLR (Test a bit and clear — testowanie bitu i zerowanie)**Składnia:**

BCLR #<dana>, <ea>

BCLR Dn, <ea>

Atrybuty:

rozmiar: B, L

Działanie:

Instrukcja BCLR testuje określony bit operandu przeznaczenia i w zależności od jego wartości, ustawia bit Z rejestru statusowego. Potem dany bit operandu zostaje wyzerowany. Numer bitu do testowania może być określony przez daną natychmiastową (testowanie statyczne) lub zawartość określonego rejestru danych (testowanie dynamiczne). W wypadku testowania operandu w pamięci, instrukcja ma rozmiar bajtu. Gdy będziemy testować bit rejestru danych, ma ona rozmiar długiego słowa.

Kody warunków:

X — nie zmieniany,

N — nie zmieniany,

Z — ustawiany, gdy testowany bit był zerowy,

V — nie zmieniany,

C — nie zmieniany.

Dozwolone tryby adresowania:

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
tak	nie	tak	tak	tak	tak	tak
X.w	X.1	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	nie	nie	nie	nie	nie

Przykład:

BCLR D0, (A0)+

Pod adresem wskazywanym przez zawartość rejestru A0, przetestowany i wyzerowany zostanie bit określony przez zawartość rejestru D0. Zawartość rejestru A0 zostanie zwiększona o \$1.

Rozkaz BSET (Test A Bit and Set—testowanie bitu i ustawianie)**Składni .:**

BSET #<dana>,<ea>

BSET Dn,<ea>

Atrybuty:

rozmiar: B, L

Działanie:

Instrukcja BSET testuje określony bit operandu przeznaczenia i w zależności od jego wartości, ustawia bit Z rejestru statusowego. Potem dany bit operandu zostaje ustawiony na 1. Numer bitu do testowania może być określony przez daną natychmiastową (testowanie statyczne) lub zawartość określonego rejestru danych (testowanie dynamiczne). W wypadku testowania operandu w pamięci, instrukcja ma rozmiar bajtu. Gdy będziemy testować bit rejestru danych, ma ona rozmiar długiego słowa.

Kody warunków:

X — nie zmieniany,

N — nie zmieniany,

Z — ustawiany, gdy testowany bit był zerowy.

V — nie zmieniany,

C — nie zmieniany.

Dozwolone tryby adresowania:

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
tak	nie	tak	tak	tak	tak	tak
X.w	X.l	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	nie	nie	nie	nie	nie

Przykład:

BSET #5, D6

Bit 5 rejestru danych D6 zostanie ustawiony na 1.

d. Poruszanie głowic

W celu poruszania głowic po dysku (przesuwane są równocześnie dwie głowice), korzystamy z rejestru \$bfd100. Za pomocą pierwszego bitu określamy kierunek przesuwu głowicy. Jeśli będzie on równy 1, to głowice będą się przesuwać w kierunku cylindra 0 (na zewnątrz). Jeśli jednak go wyzerujemy, to głowice będą się przesuwać w kierunku cylindra 79 (do środka).

Gdy określimy kierunek przesuwu głowicy, to dajemy sygnał STEP. Polega on na nadaniu bitowi 0 w rejestrze \$bfd100 kolejno następujących wartości: 1, 0, 1. Zabezpiecza to przed przypadkową zmianą tego rejestru, która mogłaby spowodować „wariację” stacji dysków.

```

Step0:  BSET    #1,$bfd100      ;kierunek ruchu głowicy na zewnątrz
        NOP
        NOP
        BCLR    #0,$bfd100      ;impuls, najpierw wysoki, krótki niski,
                                   ;potem wysoki (1, 0, 1)
        NOP
        NOP
        BSET    #0,$bfd100
        RTS

Step1:  BCLR    #1,$bfd100      ;kierunek ruchu głowicy do wewnątrz
        NOP
        NOP
        BCLR    #0,$bfd100      ;impuls, najpierw wysoki, krótki niski,
                                   ;potem wysoki (1, 0, 1)
        NOP
        NOP
        BSET    #0,$bfd100
        RTS

```

Uwaga: po przesunięciu głowicy o jedną ścieżkę, należy pamiętać, aby odczekać co najmniej 3 milisekundy przed wykonaniem jakichkolwiek operacji związanych ze stacją dysków. Nie należy także próbować przesuwania głowic poza 0 lub 159 ścieżkę.

e. Orientacja na dysku

Jak wcześniej wspomniałem, w celu zapewnienia orientacji na dysku musimy odnaleźć ścieżkę 0. W tym celu należy skorzystać z bitu 4 w rejestrze \$bfe001. Jeśli będzie on wynosił 0, to głowica znajduje się nad ścieżką zerową. Postępujemy więc w taki sposób:

- sprawdzamy czy głowica jest nad ścieżką zerową,
- jeśli nie, to przesuwamy głowicę na zewnątrz,
- skaczemy do punktu 1.

A oto przykładowa procedura:

```

Search0:
        BTST    #4,$bfe001
        BEQ.S   Track0
        BSR.S   Step0
        BSR.S   Wait          ;trzeba dać głowicy trochę czasu na
        BRA.S   Search0       ;przesunięcie
Track0:  RTS

```

```

Wait:    MOVE    #800,d0
WLoop:   NOP
        DBF     d0,WLoop
        RTS

```

f. Sprawdzanie, czy dysk jest w stacji

Aby sprawdzić czy dysk jest w stacji, musimy skorzystać z bitu CHNG w rejestrze \$bfe001. Sygnalizuje on zmianę dyskietki.

W chwili wyjęcia dyskietki ze stacji, wartość bitu CHNG wynosi 0 i pozostaje tak aż do momentu, w którym zostaje wysłany sygnał STEP. Jeśli w tym momencie w stacji pojawi się dyskietka, to bit CHNG zostanie ustawiony na 1. W przeciwnym wypadku bit CHNG pozostanie niezmieniony. W takiej sytuacji trzeba, w regularnych odstępach czasu, wysyłać sygnał STEP i sprawdzać, czy dysk jest w stacji. Te regularne impulsy STEP, są właśnie przyczyną „stukania” stacji dysków, gdy nie ma w niej żadnej dyskietki.

Oto prosty przykład. Jeśli dyskietki nie ma w stacji, to ekran zmienia kolor na czerwony. Jeśli dyskietkę włożymy do stacji, kolor zmienia się na zielony.

```

;-----
Test_Disk:
        MOVE.W   #$4000,$dff09a
        MOVE.B   #$ff,$bfd100
        BCLR     #$03,$bfd100           ;stacja df0:

Start:   MOVE.W   #50*2,D1               ;czekamy 2 sekundy
        BSR      Very_Long_Wait
        BCLR     #0,$bfd100
        BSR      Wait
        BSET     #0,$bfd100
        BSR      Wait
        BTST     #2,$bfe001
        BEQ      No_Disk

Disk:    BTST     #6,$bfe001
        BEQ      End
        MOVE.W   #$0f0,color
        BRA      Disk_Found

No_Disk: BTST     #6,$bfe001
        BEQ      End
        MOVE.W   #$f00,color
        BRA      Start

Disk_Found: BRA      Start
;-----
Very_Long_Wait: MOVE.W   #0,Timer
L_Wait:  BSR      Wait_Beam
        ADD.W    #1,Timer
        MOVE.W   Timer,D0
        CMP.W    D1,D0
        BLE      L_Wait
        MOVE.W   #$00f,$dff180
        RTS
;-----
Wait_Beam: MOVE.L   $dff004,d0
          ASR.L    #8,D0
          ANDI.L   #$1ff,d0
          MOVE.W   Color,$dff180
          CMP.W    #$12d,d0
          BNE      Wait_Beam
          RTS

```

```

;-----
End:          BSR      Offs
              MOVE.W  #$c000,$dff09a
              RTS
;-----
Wait:         NOP
              NOP
              NOP
              RTS
;-----
Offs:         BSET     #$03,$bfd100
              BSR      Wait
              BSET     #$07,$bfd100
              BSR      Wait
              BCLR     #$03,$bfd100
              BSR      Wait
              BSET     #$03,$bfd100
              RTS
;-----
Timer:        DC.W    0
Color:        DC.W    0
;-----

```

g. Sprawdzanie zabezpieczenia dysku

Jeśli będziemy chcieli coś nagrać na dyskietce, to najpierw należy sprawdzić czy nie jest ona zabezpieczona przed zapisem.

W tym celu musimy sprawdzić bit WPRO w rejestrze \$bfe001. Jeśli będzie on równy 1, to dysk jest niezabezpieczony. Jednak zanim sprawdzisz zabezpieczenie, to upewnij się że dysk jest w stacji. Inaczej bit WPRO będzie cały czas równy 0.

```

WriteProtect:
    BTST     #3,$bfe001
    BNE.S    NotProtected
Protected:

```

h. Wyłączanie silnika stacji dysków

Po skończonych operacjach dyskowych, musimy wyłączyć silnik stacji dysków. Jest to operacja odwrotna do włączania. Oto przykład.

```

Df00ff: BSET     #3,$bfd100          ;nieaktywna stacja 0
        NOP
        NOP
        BSET     #7,$bfd100          ;wyłączenie silnika
        NOP
        NOP
        BCLR     #3,$bfd100          ;aktywna stacja 0
        RTS

```

9.6.5.3 Użytkowanie DMA dysku

Jeśli już ustawiliśmy głowicę w żądanym miejscu dysku, to teraz musimy w jakiś sposób zmusić system do odczytania danych. Dokonujemy tego za pomocą kanałów DMA dysku. W tym celu musimy wykorzystać następujące rejestry:

- DSKPTH i DSKPTL (\$dff020 i \$dff022),
- DSKLEN (\$dff024),

- DSKSYNC (\$dff07e),
- ADKCON (\$dff09e)(w) i ADKCONR (\$dff010)(r).

Zanim uruchomimy DMA dysku, musimy upewnić się, że poprzednie DMA dysku zostało zakończone.

Najpierw określamy adres bufora w pamięci. Do rejestrów DSKPTH i DSKPTL zapisujemy adres początku pamięci, do którego mają być przesyłane dane. Następnie musimy zainicjalizować rejestr DSKLEN. Oto znaczenie bitów w tym rejestrze.

- bit 15 (DMAEN) — uaktywnienie DMA dysku i rozpoczęcie operacji zapisu/odczytu — należy jednak pamiętać, że najpierw DMA dysku musi być włączony (należy ustawić bit DSKEN w DMACON (\$dff09e)),
- bit 14 (WRITE) — tryb zapisu — ustaw na 1 jeśli chcesz coś na dysku zapisać,
- bity 13 – 0 (LENGTH) — liczba słów do przesłania.

Uwaga: należy pamiętać, że rejestr DSKLEN musimy zapisać dwa razy pod rząd, tą samą wartością. Dopiero wtedy DMA dysku zostanie uruchomione i rozpocznie się przesyłanie danych.

Tak więc sekwencja inicjalizacji DMA dysku, powinna wyglądać następująco:

- Zapisanie 0 do DSKLEN, aby wyłączyć DMAEN.
- Jeśli nie jest dotychczas ustawiony bit DSKEN w rejestrze DMACON, to należy go wyłączyć.
- Zapisać do rejestrów DSKPTH i DSKPTL adres bufora.
- Zapisać do DSKLEN odpowiednią wartość LENGTH i WRITE, włącznie z ustawieniem bitu DMAEN.
- Ponownie zapisać tę samą wartość do DSKLEN (to spowoduje rozpoczęcie przesyłania danych).
- Poczekać, aż DMA dysku zostanie wykonane.
- Dla bezpieczeństwa ustawić DSKLEN na \$0000.

Aby sprawdzić czy sterownik zakończył przenoszenie danych, musimy wykorzystać przerwanie DSKBLK (bit 1 w INTREQ).

Czasami jednak nie chcemy jednorazowo wczytywać do pamięci całej ścieżki. W takim przypadku można ustalić rozpoczęcie DMA dysku od wyznaczonego położenia. Aby tego dokonać należy zapisać słowo danych, od którego sterownik ma zacząć przenoszenie danych, do rejestru DSKSYNC (\$dff07e) i nakazać, za pomocą 10 bitu w ADKCON, synchronizację sterownika.

Sterownik dysku, jak zwykle po włączeniu DMA dysku, odczytuje dane z dysku, ale nie może ich zapisywać do pamięci. Zamiast tego, w sposób ciągły, porównuje on każde ze słów ze słowem w rejestrze DSKSYNC. Gdy oba słowa są takie same, rozpoczyna się przenoszenie danych, które jest już kontynuowane w zwykły sposób.

A oto znaczenie bitów 8 – 10 i 12 – 15 z rejestru ADKCON.

- bit 15 (SET/CLR) — ustawianie lub czyszczenie bitów tego rejestru (1 — bity zapisane na 1, będą ustawione; 0 — bity zapisane na 1, będą wyzerowane)
- bit 14 – 13 (PRECOMP) — czas prekompensacji dysku (%00 - żaden, %01 - 140 nanosekund, %10 - 280 nanosekund, %11 - 560 nanosekund).

- bit 12 (MFMPREC) — wybranie formatu MFM (1); wybranie formatu GCR (0).
- bit 10 (WORDSYNC) — zmuszenie kontrolera dysku do zsynchronizowania ze słowem danych zapisanym do rejestru DSKSYNC (\$dff07e).
- bit 9 (MSBSYNC) — udostępnienie synchronizacji GCR dla operacji dyskowych (1).
- bit 8 (FAST) — szybkość sterownika (1-2 μ s na bit; 0-4 μ s na bit).

9.6.5.4 Przykładowa procedura odczytu z dysku

```

Start:      MOVE.W  #$4000,$dff09a
            MOVE.W  #0,D0
            MOVE.W  #2,D1
            MOVE.L  #Bufor1,A0
            JSR     READ
            MOVE.W  #$c000,$dff09a
            RTS

;-----
;D0 - Początkowa ścieżka
;D1 - Ilość ścieżek do doczytania
;A0 - Pod jaki adres w pamięci?

Read:       MOVEM.L D0-A6,-(A7)
            SUBQ.W  #1,D1
            TST.W   D1
            BLT     End_Read
            BSR     Init_Read
Read_Loop:  BSR     Read_Track
            BSR     Decode_Track
            LEA     $1600(a0),a0      ;następny fragment pamięci
            ADDQ.W  #1,D0
            BSR     Heads_Step
            DBF     D1,Read_Loop
End_Read:   BSR     Motor_Off
            MOVEM.L (A7)+,D0-A6
            RTS

;-----
Init_Read:  MOVE.B  #$ff,$bfd100
            BSR     Wait
            BCLR    #$07,$bfd100      ;włączamy silnik
            BSR     Wait
            BCLR    #$03,$bfd100      ;stacja df0:
            BSR     Wait               ;czekamy
            BSR     Disk_Wait          ;czy DMA dysku wolne?
            BSET     #$01,$bfd100      ;kierunek ruchu głowicy: na zewnątrz

SeeTrk0:    BTST    #$04,$bfe001      ;czy głowica znajduje się na
            BEQ     Trk0Found          ;ścieżce 0?
            BCLR    #0,$bfd100        ;przesuwamy głowice
            BSR     Wait
            BSET     #0,$bfd100        ;1, 0, 1
            BSR     Timer_Wait
            BRA     SeeTrk0

;-----
Trk0Found:  BSET     #$02,$bfd100      ;wybieramy dolną głowicę
            MOVEQ   #0,D2
            MOVE.B  D0,D2
            LSR.W   #1,D2              ;doczytujemy ścieżki, nie cyl.
            BCC     Head_Sel          ;jeśli nieparzyste to wybieramy
            BCLR    #$02,$bfd100      ;górną głowicę

```

```

Head_Sel:      SUBQ.W #1,D2          ;jeśli ścieżka zerowa, to
               BLT      Powrot        ;głowica jest już ustawiona
               BCLR     #$01,$bfd100 ;kierunek ruchu do wewnątrz
Step_In:       BCLR     #$00,$bfd100 ;1, 0, 1
               BSR      Wait          ;StepIn
               BSET     #$00,$bfd100
               BSR      Timer_Wait
               DBF      D2,Step_In
               RTS

;-----
Powrot:        RTS

;-----
Disk_Wait:     BTST     #$05,$bfe001
               BNE      Disk_Wait
               RTS

;-----
Test_Disk:     BTST     #2,$bfe001
               BEQ.S    Error
               RTS
Error:         MOVE.W   $dff006,$dff180
               BRA.S    Error

;-----
Read_Track:    MOVE.W   #$8400,$dff09e ;synchronizacja danych
               MOVE.W   #$4489,$dff07e ;słowo synchronizacji
               BSR      Disk_Wait      ;czy stacja gotowa?
               MOVE.W   #$4000,$dff024 ;uaktywniamy DMA dysku
               MOVE.L   $Bufor,$dff020 ;zapisujemy adres bufora
               MOVE.W   #$9800,$dff024 ;zapisujemy długość danych w
               MOVE.W   #$9800,$dff024 ;słowach i uruchamiamy DMA
               MOVE.L   #$80000,d3
Wait_Block:    MOVE.W   $dff01e,d2      ;czy DMA już skończył?
               BTST     #$01,d2
               BNE      Block_Found    ;jeśli tak, to wyjdź
               SUBQ.W   #1,D3
               BNE      Wait_Block
               BRA      End_Track
Block_Found:   MOVE.W   D2,$dff09c
End_Track:     MOVE.W   #$4000,$dff024
               MOVE.W   #$0400,$dff09e
               RTS

;-----
Heads_Step:    BCHG     #2,$bfd100      ;raz dolna, raz górna głowica
               BTST     #0,D0           ;jeśli ścieżka nieparzysta,
               BNE      Powrot          ;to nie przesuwaj
               BCLR     #1,$bfd100      ;kierunek "IN"

               BCLR     #0,$bfd100      ;STEP dla głowic
               BSR      Wait
               BSET     #0,$bfd100      ;1, 0, 1
               BSR      Timer_Wait
               RTS

;-----
Motor_Off:     BSET     #$03,$bfd100
               BSR.W    Wait
               BSET     #$07,$bfd100
               BSR.W    Wait
               BCLR     #$03,$bfd100
               BSR.W    Wait
               BSET     #$03,$bfd100
               RTS

;-----

```

```

Wait:          NOP
               NOP
               NOP
               RTS

;-----

Timer_Wait:    ;czekamy 1/100 sekundy
               MOVE.B #$00,$bfd00 ;czyścimy CRA
               MOVE.B #$7f,$bfd00 ;czyścimy ICR
               MOVE.B #$00,$bfd400 ;timer a - młodszy bajt
               MOVE.B #$20,$bfd500 ;timer a - starszy (wartość=$2000)
               MOVE.B #$09,$bfd00  ;set oneshot/start
Wait_Timer:    BTST  #0,$bfd00     ;testujemy bit w ICR
               BEQ.S Wait_Timer
               RTS

;-----

Decode_Track:  MOVEM.L D0-A6,-(A7)
               MOVE.L #Bufor,A2
               MOVE.L A2,A5
               MOVEQ  #11-1,D6

Decode_Loop:   MOVE.W #$4489,d0
Srchr_Loop1:   CMP.W  (A5)+,D0
               BNE    Srchr_Loop1
Srchr_Loop2:   CMP.W  (A5)+,D0
               BEQ    Srchr_Loop2

               SUB.L  #$a,a5
               LEA    8(A5),A3 ;w A5 początek sektora (ścieżki)

Next_1b: MOVE.L (A3)+,D0 ;dana zakodowana w formacie MFM
               MOVE.L (A3)+,D1 ;dwa długie słowa
               ANDI.L #$55555555,d0
               ANDI.L #$55555555,d1
               ADD.L  D0,D0
               OR.L   D1,D0
               MOVE.L D0,Trk_Format

               MOVEQ  #0,D0
               MOVE.B Trk_Format+2,D0 ;numer sektora
               MOVEQ  #9,D1
               LSL.L  D1,D0 ;*512
               MOVE.L A0,A3
               ADDA.L D0,A3 ;gdzie w pamięci dekodujemy sektor
               LEA    $40(a5),a1
               MOVE.L A1,A2
               ADDA.L #512,A2
               MOVE.L #128-1,D4 ;128, bo 4*128=512

Decode_Sector:
Next_2b: MOVE.L (A2)+,D1
               ANDI.L #$55555555,d1
               MOVE.L (A1)+,D0 ;dekoduj długie słowo
               ANDI.L #$55555555,d0
               ADD.L  D0,D0
               OR.L   D0,D1
               MOVE.L D1,(A3)+ ;zapisz do pamięci
               DBF    D4,Decode_Sector
               ADDA.L #$440,a5 ;przechodzimy do następnego sektora
               DBF    D6,Decode_Loop
               MOVEM.L (A7)+,D0-A6
               RTS

Trk_Format:    DC.L    0
Bufor:         BLK.B   $3000,0 ;tutaj przechowujemy doczytaną ścieżkę

Bufor1:        BLK.B   100*1024,0 ;zarezerwowane 100 KB

```

Oto opis nowych instrukcji:

Rozkaz TST (Test an operand — testowanie operandu)

Składnia:

TST <ea>

Atrybuty:

rozmiar: B, W, L

Działanie:

Instrukcja TST testuje określony operand określony adresem efektywnym i w zależności od tego, ustawia kody warunków. Zawartość operandu nie ulega zmianie.

Kody warunków:

- X — nie zmieniany,
- N — ustawiany, gdy najstarszy bit wyniku równy jest jeden, w przeciwnym wypadku kasowany,
- Z — ustawiany, gdy wynik był zerowy, w przeciwnym wypadku kasowany,
- V — nie zmieniany,
- C — nie zmieniany,

Dozwolone tryby adresowania:

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
tak	nie	tak	tak	tak	tak	tak
X.w	X.1	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	nie	nie	nie	nie	nie

9.6.6 Bootblock

Bardzo często przy pisaniu programów zajmujących całą dyskietkę, istnieje potrzeba, by po włożeniu dyskietki, doczytała się procedura inicjująca cały program. Można tego dokonać za pomocą *bootblocku*. Są to dwa pierwsze sektory dysku, które system operacyjny odczytuje i traktuje jako program do wykonania. Jak go wykorzystać, dowiemy się za chwilę.

Oto jak wygląda struktura bootblocku:

```

DC.B    "DOS",0          ;długie słowo informujące, że jest to bootblock
                        ;z zawartym programem do wykonania
DC.L    0                ;miejsce na sumę kontrolną
DC.L    880              ;wskaźnik rootblocku

```

;właściwa procedura zawierająca loader ładujący właściwy program

Pierwsze długie słowo jest informacją dla systemu operacyjnego Amigi, że dyskietka jest dyskietką przeznaczoną dla Amigi, zawierającą bootblock z programem do wykonania.

Następne długie słowo jest miejscem przeznaczonym na *sumą kontrolną bootblocku*. Jeśli nagramy bootblock na dyskietkę, to należy wywołać w assemblerze funkcję CC (Calc Checksum), która obliczy sumę i nagra ją na dyskietce w tym miejscu.

Po włożeniu dyskietki do stacji dysków system odczytuje bootblock i sprawdza pierwsze długie słowo. Jeśli jest prawidłowe (litery "D", "O", "S" zakończone zerem), to następuje sprawdzenie sumy kontrolnej. Jeśli wartość sumy kontrolnej zgadza się, system przestępuje do wykonania procedury.

Trzecie długie słowo określa *rootblock*. Jednak jeśli nie korzystamy z systemu operacyjnego, to nie powinno nas to interesować.

Nasza procedura zaczyna się dopiero od trzynastego bajtu. Ponieważ nigdy nie ma pewności, że system operacyjny doczyta bootblock pod ten, a nie inny adres, procedura powinna być w pełni relokowalna. Oznacza to, że powinna działać w taki sam sposób, niezależnie od adresu, pod jaki zostanie doczytana. Jednak nie zawsze jest możliwe napisanie w pełni relokowalnej skomplikowanej procedury. Można ten problem rozwiązać w następujący sposób. Otóż asemblujemy procedurę od określonego adresu (na przykład \$60000) i piszemy procedurę relokowalną, która umieści ten fragment programu pod adresem, pod którym był zasemblowany. Oto jak może wyglądać taka procedura kopiująca:

```

        LEA     Datas(Pc),A0
        LEA     $60000,a1
        MOVE.W  #ile_Bajtów,D7
Copy_Loop:
        MOVE.B  (A0)+,(A1)+
        DBF     D7,Copy_Loop
        JMP     $60000

```

Pod etykietą datas powinna znajdować się właściwa procedura zasemblowana pod adres \$60000.

A tak będzie wyglądał przykładowy bootblock:

```

;-----
Start:
        DC.B    'DOS',0
        DC.L    0
        DC.L    880

        LEA     Datas(Pc),A0
        LEA     $60000,a1
        MOVE.W  #End-Datas,D7
Copy_Loop:
        MOVE.B  (A0)+,(A1)+
        DBF     D7,Copy_Loop
        JMP     $60000

Datas:  MOVE.W  #$4000,$dff09a
Loop:   MOVE.W  #$f00,$dff180
        BRA     Loop
End:
;-----

```

Aby go uruchomić, należy go zasemblować, potem nagrać na dysk, podając następujące parametry:

- >WS
- RAM PTR>Start
- DISK PTR>0
- LENGHT>2

oraz obliczyć sumę kontrolną funkcją CC:

- >CC 0

Potem wystarczy zresetować komputer i włożyć dyskietkę do stacji dysków.

Rozkaz JMP (Jump — skok)

Składnia:

JMP <ea>

Atrybuty:

bez rozmiaru

Działanie:

Instrukcja JMP powoduje kontynuowanie programu od miejsca wskazywanego przez adres efektywny. Adres jest generowany jest na podstawie obliczenia adresu efektywnego.

Kody warunków:

- X — nie zmieniany,
- N — nie zmieniany,
- Z — nie zmieniany,
- V — nie zmieniany,
- C — nie zmieniany.

Dozwolone tryby adresowania:

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
nie	nie	tak	nie	nie	tak	tak
X.w	X.1	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	tak	tak	nie	nie	nie

Przykład:

JMP (A0)

Kontynuowanie programu nastąpi od adresu wskazywanego przez zawartość rejestru A0.

9.7 Obsługa urządzeń sterowniczych

Z tego rozdziału dowiesz się jak korzystać z urządzeń sterowniczych (myszki, trackballa, joysticka).

9.7.1 Myszka i trackball

Procedura kontrolująca te dwa urządzenia, jest w zasadzie identyczna. Posiadają one dwa liczniki pozycji, znajdujące się w rejestrach JOY0DAT i JOY1DAT. Bity 15 – 8 tych rejestrów, to licznik pionowy, a 7 – 0 to licznik poziomy. Liczniki są zwiększane, gdy mysz porusza się w prawo lub w dół. Liczniki są zmniejszane, gdy mysz porusza się w lewo lub do góry.

Jeśli będziesz chciał używać myszy, to musisz odczytywać te rejestry co najmniej raz na ramkę. Aby określić kierunek i prędkość poruszania myszy, musisz odjąć bieżące wartości rejestrów, od poprzednich. Wartość bezwzględna, będzie wskazywać szybkość poruszania się myszy, a znak — kierunek.

Oto przykładowa procedura pozwalająca na sterowanie spritem po ekranie.

Uwaga: Procedura ta nie „zabija” systemu operacyjnego.

```

X_Offs_Spr      =      128-8
Y_Offs_Spr      =      44-7
High_Spr        =      16
Xmin_Spr        =      0
Ymin_Spr        =      0
Xmax_Spr        =      319
Ymax_Spr        =      255
Start:
Wait:           BSR      Init           ;inicjacja procedury
                CMP.B    #$ff,$dff006   ;czekanie na ramkę
                BNE      Wait
                BSR      Mouse_Con
                BSR      Sprite_Position
                BTST     #6,$bfe001
                BNE      Wait
                RTS

;-----
;Mouse-Control
;-----
Mouse_Con:
                MOVEM.L  D0-D1,-(A7)
                MOVE.B   $dff00a,d0
                MOVE.B   D0,D1
                SUB.B     Pozy,D0
                EXT.W     D0
                ADD.W     D0,Spry
                MOVE.B   D1,Pozy
                MOVE.B   $dff00b,d0
                MOVE.B   D0,D1
                SUB.B     Pozx,D0
                EXT.W     D0
                ADD.W     D0,Sprx
                MOVE.B   D1,Pozx
                MOVEM.L  (A7)+,D0-D1
                RTS

```



```

;-----
;Sprite Real Coordrs
;D0-X , D1-Y, A0-Sprite
;-----
Sprite_Position:
        CMP.W   #Xmin_Spr,Sprx
        BGT     X_Small
        MOVE.W   #Xmin_Spr,Sprx
X_Small:
        CMP.W   #Ymin_Spr,Spry
        BGT     Y_Small
        MOVE.W   #Ymin_Spr,Spry
Y_Small:
        CMP.W   #Xmax_Spr,Sprx
        BLT     X_Big
        MOVE.W   #Xmax_Spr,Sprx
X_Big:
        CMP.W   #Ymax_Spr,Spry
        BLT     Y_Big
        MOVE.W   #Ymax_Spr,Spry
Y_Big:
        MOVE.W   Sprx,D0
        MOVE.W   Spry,D1
        LEA      Sprite,A0

        ADDI.W   #X_Offs_Spr,D0
        ADDI.W   #Y_Offs_Spr,D1
        BTST     #0,D0
        BEQ      No_Second
        ORI.B    #1,3(A0)
        BRA      Second
No_Second:
        BCLR     #0,3(A0)
Second:
        ASR.W    #1,D0
        MOVE.B   D0,1(A0)
        CMP.W    #$ff,d1
        BLE      No_Rast_0
        ORI.B    #4,3(A0)
        BRA      Rast_0
No_Rast_0:
        BCLR     #2,3(A0)
Rast_0:
        MOVE.B   D1,(A0)
        ADD.W    #High_Spr,D1
        CMP.W    #$ff,d1
        BLE      No_Rast_1
        ORI.B    #2,3(A0)
        BRA      Rast_1
No_Rast_1:
        BCLR     #1,3(A0)
Rast_1:
        MOVE.B   D1,2(A0)
        RTS

;-----
;Init_Program
;-----
Init:
        LEA      Block,A0
        MOVE.L   #End_Sprite,D0
        MOVE.W   #$0124,d1
        MOVEQ    #13,D2
Loop:
        MOVE.W   D1,(A0)+
        ADDQ.W   #2,D1
        SWAP     D0
        MOVE.W   D0,(A0)+
        DBF      D2,Loop
        MOVE.L   #Copper,$dff080
        MOVE.B   $dff00a,pozy
        MOVE.B   $dff00b,pozx
        MOVE.L   #Sprite,D0
        MOVE.W   D0,MemL
        SWAP     D0
        MOVE.W   D0,Memh
        LEA      Planes,A0
        MOVE.L   #Screen,D0
        MOVE.W   D0,6(A0)
        SWAP     D0
        MOVE.W   D0,2(A0)
        RTS
;-----

```

```

Copper:      DC.L    $008e2c50
              DC.L    $00902cc1
              DC.L    $00920038
              DC.L    $009400d0
              DC.L    $00968020
              DC.L    $01080000
              DC.L    $010a0000
              DC.L    $01000000
              DC.L    $01020000
              DC.L    $01040000
              DC.L    $01000000
Planes:      DC.L    $00e00000
              DC.L    $00e20000
              DC.L    $01800000
              DC.L    $01820000
              DC.L    $01a20f80
              DC.L    $01a40f00
              DC.L    $01a60ff0
              DC.W    $0120
MemH:         DC.W    $0000
              DC.W    $0122
MemL:         DC.W    $0000
Block:        BLK.L    $e,$0
              DC.L    $01001200
              DC.L    $fffffffe

Sprite:       DC.B    0,0,0,0
              DC.W    %0000000000000000,%0000000000000000
              DC.W    %0000000010000000,%0000000010000000
              DC.W    %0000000010000000,%0000000000000000
              DC.W    %0000000010000000,%0000000010000000
              DC.W    %0000000010000000,%0000000000000000
              DC.W    %0000000010000000,%0000000010000000
              DC.W    %0000000011000000,%0000000000000000
              DC.W    %0111110111111100,%0101010001010100
              DC.W    %0000000111000000,%0000000000000000
              DC.W    %0000000010000000,%0000000010000000
              DC.W    %0000000010000000,%0000000000000000
              DC.W    %0000000010000000,%0000000010000000
              DC.W    %0000000010000000,%0000000000000000
              DC.W    %0000000010000000,%0000000000000000
              DC.W    %0000000000000000,%0000000000000000
              DC.W    %0000000000000000,%0000000000000000
End_Sprite:   DC.L    0

Sprx:         DC.W    0
Spry:         DC.W    0
Pozy:         DC.B    0
Pozx:         DC.B    0

Screen:       BLK.B    40*256,0

```

Rozkaz EXT (Sign Extend — rozszerzenie znakowe)

Składnia:

EXT Dn

Atrybuty:

rozmiar: W, L

Działanie:

Instrukcja EXT rozszerza bit znaku bajtu do słowa lub bit znaku słowa do długiego słowa. W wypadku rozszerzenia danej do rozmiaru słowa, bit 7 jest kopiowany do bitów 15 – 8 rejestru danych. Natomiast, gdy rozszerzymy słowo do długiego słowa, to bit 15 jest kopiowany do bitów 31 – 16 rejestru. Jest to tak zwane rozszerzenie znakowe,

ponieważ dzięki kopiowaniu bitu znaku, znak rozszerzonej liczby nie zmienia się. Jeśli jednak będziemy mieli pewność, że dana jest dodatnia lub chcemy zaniedbać znak danej, a zależy nam na rozszerzeniu jej do na przykład długiego słowa, to bardzo pomocna może być tutaj instrukcja ANDI.

Kody warunków:

- X — nie zmieniany,
- N — ustawiany gdy wynik operacji jest ujemny, w przeciwnym wypadku zerowany,
- Z — ustawiany gdy wynik jest zerowy, w przeciwnym wypadku zerowany,
- V — zawsze zerowany,
- C — zawsze zerowany.

Dozwolone tryby adresowania:

jedynie bezpośrednie adresowanie rejestru danych.

Przykład:

```
DIVS    D1,D0
EXT.L    D0
```

W wyniku zastosowania instrukcji EXT, rejestr D0 zawiera długie słowo wyniku (starsze słowo rejestru nie posiada już reszty z dzielenia).

Aby sprawdzić, czy któryś z przycisków jest wciśnięty, czy też nie, należy sprawdzić:

- dla lewego przycisku — bity 6 i 7 z rejestru \$bfe001 (0—naciśnięty),
- dla prawego przycisku — bit 10 rejestru \$dff016,
- dla środkowego przycisku — bit 8 rejestru \$dff016.

9.7.2 Joystick

Joystick posiada 4 jednokierunkowe przełączniki. Każdy z nich może być oddzielnie zaktywowany przez drążek. W przypadku skierowania drążka ukośnie, aktywowane są dwa przełączniki. Tak więc joystickiem możemy kierować w ośmiu kierunkach.

Jeśli któryś z przełączników jest aktywny, to wysyła sygnał 0. Wobec tego musimy odpowiednio interpretować dane pobrane z rejestrów JOY0DAT i JOY1DAT (joystick korzysta z tych samych rejestrów, co myszka). Aby otrzymać kierunki w których aktualnie skierowany jest joystick, musimy postąpić według następującego schematu (1 oznacza, że dany przełącznik jest przyciśnięty):

- stan prawego przełącznika określa bit 1 z rejestru JOYxDAT,
- stan lewego przełącznika określa bit 9 z rejestru JOYxDAT,
- aby otrzymać stan przełącznika „w dół” musisz obliczyć XOR bitów 0 i 1,
- aby otrzymać stan przełącznika „do góry” musisz obliczyć XOR bitów 9 i 8

Żeby otrzymać stan przełącznika FIRE, musimy testować bity 7 lub 6 w rejestrze \$bfe001.

A oto przykładowa procedura odczytująca aktualny stan joysticka:

```
Joy_Con:      MOVEM.L D0-D1, -(A7)

              MOVE.B #0, Gora
              MOVE.B #0, Dol
              MOVE.B #0, Lewo
              MOVE.B #0, Prawo

              MOVE.W $dff00c, d0
              BTST   #1, D0
              BEQ    E_Prawo
              MOVE.B #$ff, prawo

E_Prawo:      BTST   #9, D0
              BEQ    E_Lewo
              MOVE.B #$ff, lewo

E_Lewo:      MOVE.W D0, D1
              LSR.W  #1, D0
              EOR.W  D0, D1
              BTST   #0, D1
              BEQ    E_Dol
              MOVE.B #$ff, dol

E_Dol:      BTST   #8, D1
              BEQ    E_Gora
              MOVE.B #$ff, gora

E_Gora:      MOVEM.L (A7)+, D0-D1
              RTS

Gora:        DC.B   0
Dol:         DC.B   0
Lewo:        DC.B   0
Prawo:       DC.B   0
```

Jeśli któryś z bajtów oznaczonych „Lewo”, „Prawo”, itd. będzie równy 1, to joystick jest skierowany w tym kierunku.

9.7.3 Pióro świetlne

Sterowanie piórem świetlnym jest banalne. Musimy po prostu włączyć tryb pióra świetlnego (bit 3 w rejestrze BPLCON 0). Od tej pory liczniki promienia wizji, wskazywać będą aktualną pozycję pióra na ekranie.

9.8 Podstawy kontroli systemu operacyjnego

Każdy program działający pod kontrolą systemu operacyjnego musi działać zgodnie z innymi programami z nim współpracującymi. Musi się „dzielić” układami specjalizowanymi, czasem mikroprocesora. Aby tego dokonać, większość operacji należy wykonywać za pomocą systemu operacyjnego. Na przykład, jeśli chcemy używać blittera, to musimy odczekać aż *system operacyjny* (czyli *Kickstart*) poinformuje nas, że możemy to zrobić. System operacyjny interpretuje dane pobrane z portów, zapewnia ich kompleksową obsługę, itd. Tak więc jesteśmy uwolnieni od inicjacji copperlisty, czekania na ramkę.

Komunikacja z systemem operacyjnym odbywa się za pomocą zawartych w nim procedur. Dzięki temu, że ściśle one ze sobą współpracują, bez problemu możemy otworzyć okienko z gadżetami, itp. Nasza rola będzie się ograniczała tylko do zdefiniowania struktury okna i wywołaniu odpowiedniej procedury systemu operacyjnego. Gdy okno będzie przemieszczane, zadba o to sam system operacyjny.

Ten rozdział zawiera minimalny zakres wiedzy o systemie operacyjnym i przedstawia *podstawy* obsługi procedur systemowych. Jego dokładne omówienie we wszystkich wersjach to materiał na bardzo, bardzo grubą książkę.

9.8.1 Biblioteki

W celu efektywniejszego wykorzystania procedur systemu operacyjnego, procedury pogrupowano tematycznie i połączono w *biblioteki*. Biblioteki możemy podzielić na te, które znajdują się już na stałe w pamięci ROM, i takie, które są w razie potrzeby doczytywane z dysku. Na przykład na dysku systemowym umieszczone są:

- DISKFONT.LIBRARY,
- ICON.LIBRARY,
- INFO.LIBRARY,
- MATHIEEDUBBAS.LIBRARY,
- MATHIEEDUBTRANS.LIBRARY,
- TRANSLATOR.LIBRARY,
- VERSION.LIBRARY.

Nie sposób wymienić wszystkich bibliotek dyskowych, ponieważ istnieje możliwość tworzenia własnych. Wykorzystują to skrupulatnie przede wszystkim producenci oprogramowania, a także różnego rodzaju zapaleńcy. Do najbardziej znanych bibliotek dyskowych należą:

- REQ.LIBRARY,
- ARP.LIBRARY,
- POWERPACKER.LIBRARY.

W pamięci ROM znajdują się między innymi następujące biblioteki:

- INTUITION.LIBRARY,
- GRAPHICS.LIBRARY,
- CONSOLE.LIBRARY,
- LAYERS.LIBRARY,
- POTGO.LIBRARY,
- TIMER.LIBRARY,
- EXEC.LIBRARY,
- DOS.LIBRARY.

9.8.2 Otwieranie i zamykanie bibliotek

Aby skorzystać z procedur zawartych w bibliotece, musimy ją najpierw otworzyć. Lecz zanim do tego dojdziemy, musimy jeszcze poznać kilka istotnych rzeczy.

Dla ułatwienia korzystania z procedur systemu operacyjnego, utworzono *tabelę skoków*. System operacyjny Amigi podaje nam adres końca tej tabeli pod adresem \$4. Jest to jedyny adres, który w systemie operacyjnym nie zmienia swego znaczenia. Pozostałe obszary pamięci mogą być swobodnie wykorzystywane zarówno przez system operacyjny, jak i programy z nim współpracujące.

Tabela skoków nie ma określonego adresu i system lokuje ją tam, gdzie jest wolne miejsce.

Tak więc jeśli chcemy wywołać jakąś bibliotekę, to musimy wywołać procedurę Open Library z biblioteki EXEC. Należy pamiętać o tym, że biblioteka EXEC jest zawsze otwarta.

Procedura Open Library znajduje się pod adresem -552 względem końca tabeli skoków. W rejestrze A1 podajemy początek adresu zawierającego nazwę biblioteki, zakończoną zerem, a w D0 numer wersji. Jeśli jest on nieistotny, to wpisujemy zero. A oto jak będzie wyglądać kompletna procedura otwarcia biblioteki DOS.LIBRARY.

```

Execbase      =      4
Openlib       =     -552

Start:
        MOVE.L   Execbase,A6
        LEA      Dosname,A1
        MOVEQ    #0,D0
        JSR      OpenLib(A6)
        TST.L    D0
        BEQ      Error
        MOVE.L   D0,DosBase
        ...
Error:
        RTS

Dosname:      DC.B   "dos.library",0
              even

Dosbase:      DC.L   0

```

Nazwa biblioteki musi być podana małymi literami, zakończona zerem. System informuje nas o tym, czy udało mu się otworzyć bibliotekę, za pomocą rejestru D0. Jeśli jest on równy 0, to wystąpił błąd. W przeciwnym razie zawartość rejestru D0 należy traktować jako adres końcowy tabeli skoków danej biblioteki.

Jeśli już nie musimy korzystać z danej biblioteki, lub nasz program zakończył działanie, wtedy naszym obowiązkiem jest zamknąć bibliotekę. Dokonujemy tego za pomocą procedury CloseLibary znajdującej się pod adresem -414. W rejestrze A1 podajemy adres bazowy danej biblioteki. Oto przykład zamykający bibliotekę DOS.LIBRARY:

```

CloseLib      =     -414
        ...
        MOVE.L   Execbase,A6
        MOVE.L   Dosbase,A1
        JSR      CloseLib(A6)
        ...
        RTS

```

9.8.3 Rezerwacja pamięci

Wcześniej wspominałem, że system operacyjny Amigi dynamicznie obsługuje pamięć. Aby wykorzystać jakiś obszar pamięci, musimy ją najpierw zaalokować. Inaczej mówiąc, jest to „prośba” do systemu o przydzielenie pamięci potrzebnej na przykład do wyświetlenia obrazka. Dzięki temu będziemy mieli pewność, że nikt nam jej niespodziewanie nie zapaskudzi i będzie ona tylko i wyłącznie do naszej dyspozycji.

Rezerwacja pamięci za pośrednictwem systemu operacyjnego umożliwia bezproblemową pracę kilku programów w multitaskingu, przy czym żaden z programów nie będzie kolidował z innym.

W bibliotece EXEC.LIBRARY znajdują się procedury umożliwiające alokację obszarów pamięci. Są to AllocMem (–198) i AllocAbs (–204).

AllocMem

Wywołanie procedury AllocMem wymaga podania dwóch parametrów:

- w rejestrze D0 podajemy długość rezerwowanej pamięci w bajtach,
- w rejestrze D1 określamy rodzaj rezerwowanej pamięci.

Po zakończeniu rezerwacji, procedura zwraca nam w rejestrze D0 pewną wartość. Jeśli będzie ona równa zero, to znaczy, że wystąpił błąd. Jego przyczyną może być brak wolnej pamięci. Jeśli jednak będzie ona różna od zera, to należy ją traktować jako adres początku rezerwowanego obszaru.

A oto jak możemy określić rodzaj rezerwowanej pamięci.

- \$00001 — zarezerwowany obszar nie może być przesuwany,
- \$00002 — obszar będzie leżał w CHIP RAM,
- \$00004 — obszar będzie leżał w FAST RAM,
- \$10001 — zarezerwowany obszar nie może być przesuwany i zostanie wyzerowany,
- \$10002 — obszar będzie leżał w CHIP RAM i zostanie wyzerowany,
- \$10004 — obszar będzie leżał w FAST RAM i zostanie wyzerowany,
- \$20000 — otrzymamy największy dostępny blok pamięci.

Oto przykład krótkiej procedury rezerwującej pamięć.

```
ExecBase      =      4
AllocMem      =     -198

...
MOVE.L    #$5000,d0
MOVE.L    #$00002,d1
MOVE.L    ExecBase,A6
JSR       AllocMem(A6)
TST.L     D0
BEQ       Error
MOVE.L    D0,Memory_Pointer
...

Memory_Pointer: DC.L    0
```

AllocAbs

Procedura AllocAbs rezerwuje obszar pamięci o podanym przez nas początku i określonej długości. W rejestrze A1 podajemy adres początku obszaru rezerwowanej przez nas pamięci, w D0 jej długość. Jeśli procedura zwróci w rejestrze D0 zero, to alokacja nie miała miejsca.

Tak wygląda przykładowa procedura:

```
ExecBase      =      4
AllocAbs      =      -204

...
MOVE.L  #$70000,a1
MOVE.L  #$2800,d0
MOVE.L  ExecBase,A6
JSR     AllocAbs(A6)
TST.L   D0
BEQ     Error
...
```

9.8.4 Zwalnianie zarezerwowanej pamięci

Gdy zarezerwowana uprzednio pamięci nie jest nam już potrzebna, to naszym obowiązkiem jest jej zwolnienie. Dokonujemy tego za pomocą procedury FreeMem (-210). W rejestrze A1 podajemy adres zarezerwowanego obszaru, a w D0 jego długość. Jeśli po wyjściu z procedury w rejestrze D0 będzie zero, to obszar nie został zwolniony.

Uwaga: należy ostrożnie posługiwać się tą procedurą, ponieważ próba zwolnienia niezarezerwowanej pamięci, może doprowadzić do „zawieszenia się” systemu.

Przykładowy program zwalniający obszar pamięci może wyglądać tak:

```
ExecBase      =      4
FreeMem       =      -210

...
MOVE.L  #$70000,a1
MOVE.L  #$2800,d0
MOVE.L  ExecBase,A6
JSR     FreeMem(A6)
TST.L   D0
BEQ     Error
...
```

9.8.5 Włączanie i wyłączanie multitaskingu

System operacyjny Amigi pozwala na korzystanie z multitaskingu, czyli uruchomienie kilku programów jednocześnie. Jednak często może nam zależeć, by działał tylko nasz program, a wielozadaniowość była wyłączona.

W bibliotece EXEC umieszczone są dwie procedury: Forbid (-132) i Permit (-138). Wywołanie procedury Forbid wyłącza multitasking, natomiast Permit go włącza.

9.8.6 Operacje na plikach

W celu komunikacji ze stacją dysków (a także innymi urządzeniami), stworzono bibliotekę DOS.LIBRARY. Umożliwia ona wszelkiego rodzaju operacje na plikach (usuwanie z dysku, doczytywanie, itp.).

9.8.6.1 Otwarcie i zamknięcie pliku

Zanim jakkolwiek plik odczytamy lub zapiszemy na dysku, to najpierw musimy go otworzyć za pomocą procedury Open (-30). W rejestrze D1 podajemy adres nazwy pliku zakończonej zerem, a w D2 — rodzaj otwarcia. Jeśli zapiszemy do D2 wartość #1005, to poinformujemy system, że chodzi o plik już istniejący, natomiast wartość #1006 oznacza, że chcemy stworzyć nowy plik.

Procedura systemowa zwraca w rejestrze D0 pewną wartość — *filehandle*. Będzie ona używana przy zapisie lub odczycie, do rozróżnienia otwartych plików. Dlatego też należy ją przechować.

Po zakończeniu jakiegokolwiek operacji na otwartym pliku, musimy go zamknąć. Służy do tego procedura Close (-36). W rejestrze D1 musimy podać filehandle pliku, który chcemy zamknąć.

Poniższy przykład prezentuje otwarcie i zamknięcie pliku. Należy jednak pamiętać, by przedtem otworzyć bibliotekę DOS.LIBRARY i bazę tej biblioteki przechować pod etykietą DosBase.

```

Open      =      -30
Close     =      -36

...
...
...
MOVE.L    #1005,D2
JSR       Open_File
...
...
...
JSR       Close_File
...
...
...
RTS

Open_File:
MOVE.L    Dosbase,A6
MOVE.L    #Nazwa_Pliku,D1
JSR       Open(A6)
MOVE.L    D0,FileHandle
RTS

Close_File:
MOVE.L    Dosbase,A6
MOVE.L    FileHandle,D1
JSR       Close(A6)
RTS

FileHandle:    DC.L    0
Nazwa_Pliku:   DC.B    "Df0:plik1"
```

9.8.6.2 Odczyt i zapis danych

Jeśli umiemy już otwierać i zamykać pliki, to nic nie stoi na przeszkodzie, abyśmy odczytali zawartość pliku czy też utworzyli nowy. W tym celu będziemy musieli się posłużyć dwoma nowymi procedurami:

- Read (-42),
- Write (-48).

Najpierw zajmijmy się zapisem. Procedurze Write musimy podać następujące parametry:

- w rejestrze D1 filehandle pliku,
- w rejestrze D2 adres początku danych, które mają być zapisane na dysku,
- w rejestrze D3 długość danych w bajtach.

Postępujemy więc w taki sposób:

- otwieramy nowy plik i przechowujemy filehandle,
- wywołujemy procedurę Write z odpowiednimi parametrami,
- zamykamy plik.

Należy pamiętać, że biblioteka DOS.LIBRARY musi być przez cały czas otwarta!

Oto przykładowa procedura wywołująca funkcję Write:

```
Write    =          -48
        ...
        ...
        ...
Write_Binary:
        MOVE.L    Dosbase,A6
        MOVE.L    FileHandle,D1
        JSR       Write(A6)
        RTS
```

Zanim wywołasz etykietę Write_Binary, to w rejestrze D2 umieść adres początku danych, a w D3 długość danej w bajtach.

Teraz kolej na odczyt danych. Parametry są identyczne do stosowanych przy procedurze Write. Jednak nie wiemy, ile wynosi długość pliku. Wyjściem z tej sytuacji jest zapisanie do rejestru D3 wartości największej z możliwych. Wtedy cały plik zostanie doczytany do pamięci, a procedura zwróci nam w rejestrze D0 ilość doczytanych bajtów.

```
Read     =          -42
        ...
        ...
        ...
Read_Binary:
        MOVE.L    Dosbase,A6
        MOVE.L    FileHandle,D1
        JSR       Read(A6)
        RTS
```

9.9 Współpraca z systemem operacyjnym

Dotychczas nasze procedury (oprócz inicjującej odczyt klawiatury i przerwanie VERTB) zupełnie ignorowały system operacyjny. Gdybyś uruchomił na raz, zegarek, program muzyczny i assembler to prawdopodobnie, mało który z tych programów działałby prawidłowo. W tym rozdziale powiem co należy zrobić, by programy działały prawidłowo z systemem operacyjnym, a po ich zakończeniu system pozostawał nie zmieniony.

Uwaga: wszystkie informacje zawarte w tym rozdziale dotyczą programów uruchamiających się i powracające do okna CLI lub Workbench. W sytuacji gdy nie korzystasz z systemu operacyjnego, można prawie wszystkie z podanych niżej operacji pominąć.

W rozdziale tym wykorzystywane są procedury systemu operacyjnego, radzę więc najpierw przeczytać rozdział poświęcony systemowi operacyjnemu Amigi.

9.9.1 Układy CIA i przerwania

System operacyjny wykorzystuje do swoich celów zarówno przerwania, jak i układy CIA. Musimy więc zapamiętać na początku programu stary stan przerwania oraz przechować wartość wektorów przerwania. Niestety nie możemy zapamiętać stanu przerwania układów CIA. Tak więc, przy wyjściu jesteśmy zmuszeni zezwolić na wszystkie przerwania układu CIA. W miarę możliwości powinniśmy zachować stan kanałów DMA (które z nich były włączone, a które nie) i pozostałych rejestrów przerwania.

Zanim jednak zrobimy to wszystko, to najpierw musimy odłożyć na stos rejestry i wywołać procedurę systemową, powodującą przerwanie działania wszystkich aktualnie działających programów (czyli wyłączenie multitasking). Na końcu programu musimy rejestry „zdziać” ze stosu i z powrotem włączyć multitasking.

```
MOVE.L 4,A6
JSR    -132(A6)

MOVE.W $dff01c,d0
OR.W   #$8000,d0
MOVE.W D0,Old_Intena
```

```
Old_Intena:    DC.W    0
```

A oto jak może wyglądać powrót do systemu:

```
MOVE.W Old_Intena,$dff09a
MOVE.L 4,A6
JSR    -138(A6)

RTS
```

9.9.2 Rejestr D0

Przy wyjściu z naszego programu, musimy także pamiętać o wyczyszczeniu rejestru danych D0. Zwykle programy działające pod kontrolą systemu operacyjnego, informują Amiga DOS za pomocą tego rejestru, że przy obsłudze programu wystąpił jakiś błąd.

9.9.3 Copperlista systemowa

Na początku programu, jeśli będziemy chcieli wrócić z powrotem do Workbench lub CLI, musimy **koniecznie** zapamiętać copperlistę systemu operacyjnego. W tym celu musimy na początku programu odwołać się do procedur systemu operacyjnego, i przechować adres copperlisty.

```
MOVE.L 4.w,A6
MOVE.L 156(A6),A1
MOVE.L 38(A1),Old_Copper
MOVE.L #Copper,$dff080
```

```
Old_Copper:
DC.L 0
```

Na końcu programu, wpisujemy adres starej copperlisty z powrotem do rejestru \$dff080.

9.9.4 Pamięć

Nigdy, absolutnie nigdy, nie możemy pisać, ani uruchamiać programów, które z góry zakładają, że pod tym, czy tamtym adresem jest wolna pamięć. Możemy natomiast ten problem rozwiązać na dwa sposoby:

- przydzielić pamięć za pomocą odpowiednich procedur systemowych,
- stworzyć w programie odpowiednie **hunki**.

Czym jest hunk? Inaczej mówiąc, jest to taki fragment danych w programie, który stanowi wskazówkę dla systemu operacyjnego, o sposobie rozlokowania programu i jego danych w pamięci.

Hunk typu BSS

Dotychczasowe przykłady, które rezerwowały miejsce na pamięć ekranu (za pomocą funkcji **BLK.X**), wypełniały ten obszar zerami. Jeśli jednak nagralibyśmy je na dysk jako plik wykonywalne (opcja Write Object w TRASH'M ONE), to ten fragment pamięci wypełniony zerami, niepotrzebnie zajmowałby miejsce na dysku. Możemy tego uniknąć w taki sposób:

```
Section Screen_Data,BSS_C
Screen: DS.B $2800 ;obszar samych zer
```

Dzięki temu, tak nagrany program będzie zajmował znacznie mniej miejsca na dysku, a system przy doczytywaniu pliku wykonywalnego, sam rozpozna nagłówki (hunka) i przydzieli sekcji programu wymaganą pamięć.

Funkcja DS.x yyy tworzy obszar wypełniony samymi zerami, o długości yyy bajtów, słów lub długich słów (zależnie od ustawienia x na B, W lub L).

Section nazwa,BSS_x tworzy hunk BSS o nazwie nazwa i umieszcza sekcję programu w pamięci typu CHIP (BSS_C), FAST (BSS_F) lub dowolnej (BSS_P).

Hunk typu DATA

Jeśli w programie są dane, które muszą być umieszczone w pamięci typu CHIP, FAST lub aktualnie wolnej, to musisz użyć nagłówka typu DATA. Oto przykład:

```

      Section Copper_Data,DATA_C
Copper: DC.L    $00960020
      .
      .
      DC.L    $fffffffe

```

Section nazwa,DATA_x tworzy hunk DATA o nazwie nazwa i umieszcza sekcję (traktowaną jako dane) w pamięci typu CHIP (DATA_C), FAST (DATA_F) lub aktualnie dostępnej (DATA_P).

Hunk typu CODE

Aby odizolować kod programu od danych i umieścić go w określonym typie pamięci, istnieje trzeci nagłówek — CODE. Oto przykład:

```

      Section Main_Code,CODE_P
Start:      ;i tutaj umieszczony kod programu

```

Section nazwa,CODE_x tworzy hunk CODE o nazwie nazwa i umieszcza sekcję (traktowaną jako kod) w pamięci typu CHIP (CODE_C), FAST (CODE_F) lub aktualnie dostępnej (CODE_P).

Używanie kilku hunków na raz

Oto przykładowy schemat programu korzystającego z hunków.

```

;-----
      Section Główny_Kod,Code_P
;główny kod programu

      Section Obrazek,Data_C
;obrazek do wyświetlenia

      Section Wektorówka,Bss_C
;pamięć wyświetlania, zarezerwowana dla "wektorówki"

      Section Podprogramy,CODE_F
;jakieś podprogramy

      Section ,Bss_F
;pamięć zarezerwowana do jakichś przeliczeń
;-----

```

W programie można kilkakrotnie używać tych samych hunków. Jednak niektóre asemblery nie potrafią stworzyć kilku hunków tego samego typu.

Rozdział X

Grafika przestrzenna

10.1 Wstęp

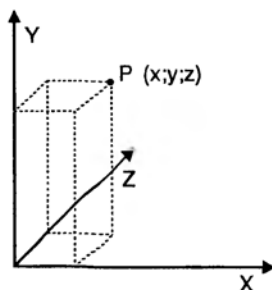
Amiga posiada wspaniałe możliwości tworzenia szybkich, trójwymiarowych animacji w czasie rzeczywistym. Zawdzięcza to przede wszystkim blitterowi, który jest w stanie bardzo szybko czyścić pamięć, wypełniać, czy też rysować linie.

Z tego rozdziału dowiesz się podstawowych rzeczy na temat programowania grafiki wektorowej (przestrzennej) na Amidze.

Uwaga: większość przykładów z tego rozdziału jest wolna w porównaniu z innymi procedurami „wektorówki”. Niestety jest to cena ich przejrzystości. Aby je nieco przyspieszyć, należy do maksimum wykorzystać rejestry i zlikwidować skoki w pętlach (podprogramy przenosić do głównej pętli, pamiętając o skasowaniu rozkazu RTS).

10.2 Trzy wymiary

Do przedstawienia jednego punktu w przestrzeni musimy użyć trzech współrzędnych x , y oraz z , przyjmując układ współrzędnych prostokątnych.



Jednak co należy zrobić, by dany punkt wyświetlić na ekranie monitora, którego ekran jest przecież płaski? Zastanówmy się, jak powstaje obraz jakiegoś przedmiotu.

Każdy oświetlony przedmiot odbija promienie świetlne. Dowolny punkt P przedmiotu jest widoczny dla obserwatora tylko wtedy, gdy promień świetlny wychodzący z punktu P trafia do jego oka. Promień taki nosi nazwę promienia widzialnego punktu P.

Jeśli teraz wyobrazimy sobie, że pomiędzy okiem obserwatora a przedmiotem została umieszczona pewna pionowa płaszczyzna, to każdy promień wychodzący z dowolnego punktu P przedmiotu przebiega ją w punkcie P'.

W taki sposób odbywa się rzutowanie przedmiotu na płaszczyznę. Takie odwzorowanie nosi nazwę rzutu środkowego lub perspektywy.

10.2.1 Przekształcenia współrzędnych w przestrzeni

Rozważmy taką sytuację: chcemy na ekranie przedstawić obraz sześcianu w perspektywie, dający się obracać i przesuwac.

Zdefiniowanie sześcianu nje powinno sprawić większych trudności. Ponieważ nie jest możliwe zapamiętanie i przedstawienie każdego jego punktu na ekranie, musimy nieco uprościć model. Wystarczy, że zapamiętamy wszystkie wierzchołki. W ten sposób otrzymujemy 8 trójek liczb:

P1(50;50;50)	P5(50;50;-50)
P2(-50;50;50)	P6(-50;50;-50)
P3(-50;-50;50)	P7(-50;-50;-50)
P4(50;-50;50)	P8(50;-50;-50)

Jak widać, sześcián początkowo znajduje się w miejscu zajmowanym przez obserwatora (współrzędne (0;0;0)). W tym położeniu może być on obracany i poddawany przesunięcióm w dowolne miejsce przestrzeni.

Jeśli mamy już współrzędne sześcianu, to możemy się pokusić o ich transformacje.

10.2.1.1 Obroty

W celu obrócenia współrzędnej wokół dowolnej osi o dany kąt, należy skorzystać z niżej podanych wzorów. Ich wyprowadzenia nie będę podawać — można je znaleźć w dowolnym podręczniku geometrii analitycznej i w większości książek poświęconych grafice komputerowej.

Obrót punktu P(x;y;z) wokół osi X o kąt α możemy opisać równaniami:

$$\begin{aligned}x' &= x \\y' &= y \cdot \cos(\alpha) - z \cdot \sin(\alpha) \\z' &= y \cdot \sin(\alpha) + z \cdot \cos(\alpha)\end{aligned}$$

Po wykonaniu tych przekształceń, otrzymamy punkt P(x';y';z'). Jak można zauważyć, punkt nie zmieni on współrzędnej x.

Analogicznie możemy opisać obroty wokół pozostałych osi: Y i Z.

$$x' = x \cdot \cos(\beta) - z \cdot \sin(\beta)$$

$$y' = y$$

$$z' = x \cdot \sin(\beta) + z \cdot \cos(\beta)$$

$$x' = x \cdot \cos(\gamma) - y \cdot \sin(\gamma)$$

$$y' = x \cdot \sin(\gamma) + y \cdot \cos(\gamma)$$

$$z' = z$$

10.2.1.2 Przesunięcia

Pomimo dokonywanych obrotów, bryła pozostanie nadal niewidoczna, ponieważ jej środek ciągle leży w punkcie początkowym układu współrzędnych, w którym także znajduje się obserwator. Aby można ją było zobaczyć musimy przesunąć równolegle o wektor $[mx;my;mz]$.

Przesunięcie nie powinno stanowić żadnego problemu. Wystarczy po prostu dodać wartości składowe przesunięcia dla poszczególnych współrzędnych:

$$x' = x + mx$$

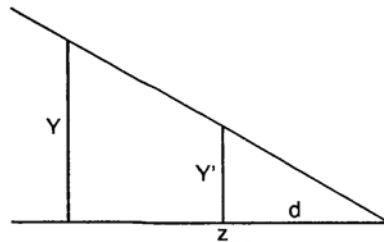
$$y' = y + my$$

$$z' = z + mz$$

10.2.2 Rzutowanie perspektywiczne

Gdy znamy już współrzędne danego obiektu (po obrotach i przesunięciach), musimy dokonać operacji rzutowania. Dzięki niej, dla każdego punktu $P(x,y,z)$ otrzymamy punkt P' o współrzędnych (x',y') będącym rzutem perspektywnym punktu P . Współrzędne punktu P' będą odpowiadać współrzędnym ekranu.

Do wyznaczenia współrzędnych rzutu wykorzystamy twierdzenie Talesa.



$$y' = \frac{y}{z} \cdot d$$

$$x' = \frac{x}{z} \cdot d$$

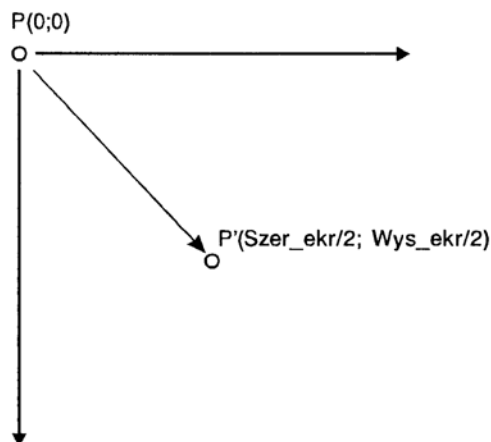
gdzie:

- d — odległość płaszczyzny rzutowania (ekranu) od obserwatora wzdłuż osi Z ,
- x, y, z — współrzędne punktu w przestrzeni,
- x', y' — współrzędne punktu na płaszczyźnie.

Jeśli teraz będziemy chcieli wyświetlić obiekt na ekranie, to nie zobaczymy go w całości, ponieważ jego środek znajduje się w punkcie $(0;0)$ układu współrzędnych ekranu (lewy górny róg). Wobec tego do współrzędnych x' i y' musimy dodać współrzędne środka ekranu. Oto wzór:

$$x'' = x' + \frac{\text{szerokość ekranu}}{2}$$

$$y'' = y' + \frac{\text{wysokość ekranu}}{2}$$



10.3 Realizacja przekształceń trójwymiarowych w assemblerze

Jak widać, do animacji wektorowej potrzebne są obliczenia na liczbach zmiennoprzecinkowych — wartości funkcji sinus i cosinus zawierają się pomiędzy 1, a -1 . Ponieważ MC 68000 nie potrafi operować na liczbach zmiennoprzecinkowych, musimy sobie poradzić w nieco inny sposób. Ale, o tym powiemy za chwilę.

Zanim w ogóle zajmiemy się jakimikolwiek przekształceniami, musimy zaplanować i stworzyć strukturę, w której będą znajdować się współrzędne wszystkich wierzchołków danego obiektu.

Nasza struktura będzie banalnie prosta. Pierwsze słowo będzie ilością punktów w tablicy minus 1, a kolejne 3 słowa — współrzędne kolejnych punktów. Weźmy wobec tego naszą przykładową kostkę:

P1(50;50;50)	P5(50;50;-50)
P2(-50;50;50)	P6(-50;50;-50)
P3(-50;-50;50)	P7(-50;-50;-50)
P4(50;-50;50)	P8(50;-50;-50)

Zapis jej struktury będzie wyglądał następująco:

```
Cube_Struct_Points:
    DC.W      8-1          ; ilość wierzchołków-1
    DC.W      50,         50,         50      ; P1
    DC.W      -50,        50,         50      ; P2
    DC.W      -50,        -50,        50      ; P3
    DC.W      50,         -50,        50      ; P4
    DC.W      50,         50,         -50     ; P5
    DC.W      -50,        50,         -50     ; P6
    DC.W      -50,        -50,        -50     ; P7
    DC.W      50,         -50,        -50     ; P8
```

Jeśli mamy już zdefiniowaną strukturę, to musimy zarezerwować w pamięci miejsce na wynikowe współrzędne dwuwymiarowe. Można to zrobić specjalną funkcją programu asemblerującego. Na przykład:

```
2D_Coords:      DS.W      512*2
```

Linia ta zarezerwuje obszar 1 kB pamięci ($2 * 512 = 1024$) na 512 współrzędnych (X;Y).

Teraz możemy napisać pętlę główną dla procedury przekształceń trójwymiarowych:

```
;-----
Przeliczanie:
    BSR      Kontrola_Obrotów          ; skocz do procedury kontrolującej
obroty
    LEA      Cube_Struct_Points,A0     ; w A0 adres struktury wierzchołków
    LEA      2D_Coords,A1              ; w A1 adres tablicy dla punktów
                                        ; wynikowych
    MOVE.W   (A0)+,D7                  ; ze struktury pobieramy ilość
                                        ; wierzchołków

Pętla:
    MOVE.W   (A0)+,D0                  ; w D0 współrzędna X
    MOVE.W   (A0)+,D1                  ; w D1 współrzędna Y
    MOVE.W   (A0)+,D2                  ; w D2 współrzędna Z

    BSR      Obroty                    ; obróć punkt

    BSR      Przesunięcie              ; przesun punkt względem początku układu

    BSR      Perspektywa              ; dokonaj rzutu perspektywicznego

    MOVE.W   D0,(A1)+                  ; zapisz współrzędne punktu do tablicy
    MOVE.W   D1,(A1)+

    DBF      D7,Pętla                  ; wykonaj wszystkie w.w. operacje na
                                        ; wszystkich wierzchołkach obiektu

    RTS
;-----
```

10.3.1 Przesunięcia

Podprogram realizujący przesunięcie będzie wyglądał następująco:

```
;-----  
;Wejście: D0-X, D1-Y, D2-Z  
;Wyjście: D0-X+MX, D1-Y+MY, D2-Z+MZ  
Przesunięcie:  
    ADD.W    MX,D0  
    ADD.W    MY,D1  
    ADD.W    MZ,D2  
    RTS  
  
MX:    DC.W    0  
MY:    DC.W    0  
MZ:    DC.W    1000
```

Przesunięcie wzdłuż osi Z musi być tak dobrane, by obiekt był widoczny dla obserwatora. Mimo, że MX i MY wynoszą zero, to zostały uwzględnione po to, by bez przeróbek można było przesuwać obiekt wzdłuż każdej osi.

10.3.2 Perspektywa

Zaprogramowanie procedury rzutującej wszystkie punkty obiektu na płaszczyznę jest proste, i sprowadza się do rozłożenia podanych wcześniej wzorów na instrukcje asemblera.

```
;-----  
;Wejście: D0-X, D1-Y, D2-Z  
;Wyjście: D0-X', D1-Y'  
  
Odległość_D    =        128  
Szr_ekr        =        320  
Wys_ekr        =        256  
  
Perspektywa:  
    EXT.L    D0  
    EXT.L    D1  
    MULS    #Odległość_D,D0  
    MULS    #Odległość_D,D1  
    DIVS    D2,D0  
    DIVS    D2,D1  
    ADD.W    #Szr_ekr/2,D0  
    ADD.W    #Wys_ekr/2,D1  
    RTS
```

Przy bardziej złożonych obiektach, czy wizualizacji „wektorowego świata”, wymagane jest, by nie dopuścić do dzielenia przez 0. W tym wypadku musimy sprawdzić, czy współrzędna Z nie wynosi przypadkiem zero. Jeśli jednak tak, to można ją zmienić na 1. Nie powinno to jednak wpłynąć na kształt obiektu.

10.3.3 Obroty

Najtrudniejszą częścią przekształceń trójwymiarowych, są obroty. Ich normalna realizacja zajmowałaby bardzo dużo czasu mikroprocesora i w rezultacie animacja byłaby bardzo wolna.

Przedstawię tutaj kilka trików pozwalających na znaczne przyspieszenie procedur obrotów.

Podstawowym problemem przy obliczaniu współrzędnych będą funkcje trygonometryczne. Ponieważ nie zależy nam na obróceniu punktu o $41^{\circ}23'11''$, możemy stabilizować wartości funkcji dla kolejnych kątów.

Niestety MC 68000 nie daje możliwości operowania na liczbach zmiennoprzecinkowych — musimy sobie poradzić w inny sposób. Weźmy może prosty przykład: chcemy pomnożyć liczbę X przez 0.01. W tym celu najpierw mnożymy ułamek przez jakąś większą liczbę (najlepiej potęgę 2). My pomnożymy przez 2048. Potem mnożymy X przez wynik. Otrzymany wynik dzielimy przez tą samą liczbę, przez którą pomnożyliśmy 0.01. Ten sposób można stosować wtedy, gdy chcemy uniknąć czasochłonnych obliczeń na liczbach zmiennoprzecinkowych.

$$0.01 \rightarrow 0.01 \cdot 2048 \rightarrow 20.28 \rightarrow X' = X \cdot 20 \rightarrow \frac{X'}{2048}$$

Tak więc przy tworzeniu naszej tablicy, wszystkie elementy musimy pomnożyć na przykład przez 256. Zapewni to nam wystarczająco dobrą dokładność.

Jak stworzyć taką tablicę? Możemy skorzystać z funkcji asemblera IS (Insert Sinus), czy CS (Create Sinus). Tworzymy etykietę `sin:` i pod nią ustawiamy kursor. Wychodzimy z edytora i wywołujemy funkcję IS. Dla tablicy Sin podajemy następujące parametry:

- `BEG>0`
- `END>360`
- `AMOUNT>256`
- `AMPLITUDE>256`
- `YOFFSET>0`
- `SIZE (B/W/L)>W`
- `MULTIPLIER>0`
- `HALF CORECTION (Y/N)>N`
- `ROUND CORECTION (Y/N)>N`

Ponieważ funkcja cosinus to to samo co sinus „przesunięty” o 90° , to parametry dla tablicy Cos będą identyczne, oprócz wartości początkowej i końcowej kąta:

- `BEG>0+90`
- `END>360+90`

Jeśli stworzyliśmy tablice funkcji sinus i cosinus dla kolejnych punktów, to czas na realizację procedury obrotów.

Na początku musimy stworzyć pomocniczą procedurę, kontrolującą wartości funkcji trygonometrycznych dla kolejnych kątów obrotu. Tak więc po wywołaniu tej procedury, słowa pod etykietami **Angle** będziemy mogli traktować jako przesunięcie względem początku tablicy. Będą one wskazywać kolejne wartości funkcji dla kolejnych kątów.

```

;-----
Kontrola_Obrotów:
    ADD.W    #2,AngleX      ;dodajemy 2, ponieważ dane w tablicy mają
                           ;długość słowa (2 bajtów)
    ADD.W    #2,AngleY
    ADD.W    #2,AngleZ
    AND.W    #$1ff,angleX
    AND.W    #$1ff,angleY
    AND.W    #$1ff,angleZ
    RTS

AngleX: DC.W    0
AngleY: DC.W    0
AngleZ: DC.W    0

```

Teraz możemy napisać już właściwą procedurę obrotów. Sprowadzi się to do przełożenia podanych wcześniej wzorów na instrukcję asemblera (wartości funkcji trygonometrycznych będziemy pobierać z tablicy).

```

;-----
;Wejście: D0-X, D1-Y, D2-Z
;Wyjście: D0-X', D1-Y', D2-Z'
Obroty:
    MOVEM.L  D3-D7/A0-A4, -(SP)

    LEA      Cos,A3
    LEA      Sin,A4

    MOVE.W   AngleX,A0
    MOVE.W   AngleY,A1
    MOVE.W   AngleZ,A2

    BSR.S    OŚ_X
    BSR.S    OŚ_Y
    BSR.S    OŚ_Z

    MOVEM.L  (SP)+,D3-D7/A0-A4
    RTS

; X'=X
; Y'=Y*cos(alfa)-Z*sin(alfa)
; Z'=Y*sin(alfa)+Z*cos(alfa)
OŚ_X:
    MOVE.W   D2,D5
    MOVE.W   D1,D4
    MULS     (A4,A0.W),D5      ;Z*sin(alfa)
    MULS     (A3,A0.W),D4      ;Y*cos(alfa)
    SUB.L    D5,D4             ;Y*cos(alfa)-Z*sin(alfa)
    ASR.L    #8,D4             ;dzielimy przez 256, ponieważ każdy element
                                ;tablicy był pomnożony przez tyle samo

    MOVE.W   D4,D6

    MOVE.W   D2,D4
    MOVE.W   D1,D5
    MULS     (A4,A0.W),D5      ;Y*sin(alfa)
    MULS     (A3,A0.W),D4      ;Z*cos(alfa)
    ADD.L    D4,D5             ;Y*sin(alfa)+Z*cos(alfa)
    ASR.L    #8,D5

    MOVE.W   D6,D2
    MOVE.W   D5,D1
    RTS

```

```

; X'=X*Cos(beta)-Z*Sin(beta)
; Y'=Y
; Z'=X*Sin(beta)+Z*Cos(beta)
OŚ_Y:
    MOVE.W    D2,D5
    MOVE.W    D0,D4
    MULS      (A4,A1.W),D5    ;Z*Sin(beta)
    MULS      (A3,A1.W),D4    ;X*Cos(beta)
    SUB.L     D5,D4           ;X*Cos(beta)-Z*Sin(beta)
    ASR.L     #8,D4

    MOVE.W    D4,D6

    MOVE.W    D2,D4
    MOVE.W    D0,D5
    MULS      (A4,A1.W),D5    ;X*Sin(beta)
    MULS      (A3,A1.W),D4    ;Z*Cos(beta)
    ADD.L     D4,D5           ;X*Sin(beta)+Z*Cos(beta)
    ASR.L     #8,D5

    MOVE.W    D6,D0
    MOVE.W    D5,D2
    RTS

; X'=X*Cos(gamma)-Y*Sin(gamma)
; Y'=X*Sin(gamma)+Y*Cos(gamma)
; Z'=Z
OŚ_Z:
    MOVE.W    D1,D5
    MOVE.W    D0,D4
    MULS      (A4,A2.W),D5    ;Y*Sin(gamma)
    MULS      (A3,A2.W),D4    ;X*Cos(gamma)
    SUB.L     D5,D4           ;X*Cos(gamma)-Y*Sin(gamma)
    ASR.L     #8,D4

    MOVE.W    D4,D6

    MOVE.W    D1,D4
    MOVE.W    D0,D5
    MULS      (A4,A2.W),D5    ;X*Sin(gamma)
    MULS      (A3,A2.W),D4    ;Y*Cos(gamma)
    ADD.L     D4,D5           ;X*Sin(gamma)+Y*Cos(gamma)
    ASR.L     #8,D5

    MOVE.W    D6,D0
    MOVE.W    D5,D1
    RTS

```

10.4 Przyspieszenie podprogramów obliczających

Pewnie zauważyłeś po złożeniu procedury przekształceń, że jest tam bardzo dużo instrukcji skoków. Przy ośmiu wierzchołkach nie będzie to zabierać zbyt dużo czasu, lecz gdy zechcemy obrócić kilkadziesiąt, to okaże się, że procedura jest dość wolna.

Aby ją nieco przyspieszyć, należy pominąć czasochłonne rozkazy: jeśli się da, zlikwidować skoki i wykorzystać wszystkie rejestry do maksimum. Metoda ta jest najprostsza z możliwych i dość uniwersalna.

```

;-----
Szr_ekr      =      320
Wys_ekr      =      256

Przeliczanie:
BSR          Kontrola_Obrotów          ;skocz do procedury kontrolującej
obroty

        MOVE.W  AngleX,A0
        MOVE.W  AngleY,A1
        MOVE.W  AngleZ,A2

        LEA     Cube_Struct_Points,A5   ;w A5 adres struktury wierzchołków
        LEA     2D_Coords,A6           ;w A6 adres tablicy dla punktów
                                           ;wynikowych

        LEA     Cos,A3
        LEA     Sin,A4

        MOVE.W  (A5)+,D7                ;ze struktury pobieramy ilość
                                           ;wierzchołków

Pętla:
        MOVE.W  (A5)+,D0                ;w D0 współrzędna X
        MOVE.W  (A5)+,D1                ;w D1 współrzędna Y
        MOVE.W  (A5)+,D2                ;w D2 współrzędna Z

        MOVE.W  D2,D3
        MOVE.W  D1,D4
        MULS    (A4,A0.W),D3
        MULS    (A3,A0.W),D4
        SUB.L   D3,D4
        ASR.L   #8,D4

        MOVE.W  D4,D5                  ;obracamy wokół osi X

        MOVE.W  D2,D4
        MOVE.W  D1,D3
        MULS    (A4,A0.W),D3
        MULS    (A3,A0.W),D4
        ADD.L   D4,D3
        ASR.L   #8,D3

        MOVE.W  D5,D2
        MOVE.W  D3,D1

        ADD.W    MZ,D2                  ;przesunięcie wzdłuż osi Z

        EXT.L   D0
        EXT.L   D1
        ASL     #8,D0
        ASL     #8,D1
        DIVS    D2,D0
        DIVS    D2,D1
        ADD.W    #Szr_ekr/2,D0
        ADD.W    #Wys_ekr/2,D1

        MOVE.W  D0,(A6)+                ;zapisz współrzędne punktu do tablicy
        MOVE.W  D1,(A6)+

        DBF     D7,Pętla

        RTS

MZ:      DC.W    1000
;-----

```

Jednak i ta procedura po pewnym czasie może okazać się za wolna. Koderzy robią więc co mogą i tablicują co się da, by wykonywać jak najmniej operacji.

Teraz, gdy mamy już skompletowane procedury przekształceń, to naszym następnym krokiem będzie stworzenie procedury rysującej wszystkie punkty obiektu na ekranie.

10.5 Wizualizacja obiektów

W tym rozdziale zajmiemy się wizualizacją trójwymiarową obiektów. Przedstawione zostaną metody reprezentacji:

- punktowej,
- szkieletowej,
- szkieletowej z usuniętymi niewidocznymi liniami,
- wypełnianej.

10.5.1 Reprezentacja punktowa

Metoda punktowa jest najprostsza ze wszystkich możliwych. Wystarczy, że pobieramy z tablicy współrzędne wierzchołków i ustawiamy punkt w odpowiednim miejscu ekranu.

Pod etykietą **Plane** powinien znajdować się adres ekranu na którym chcesz narysować obiekt.

```
Dot_Set:
    MOVE.W   Cube_Struct_Points, D7
    LEA      2D_Coords, A0
    MOVE.L   Plane, A1
Set_Loop:
    MOVE.W   (A0)+, D0
    MOVE.W   (A0)+, D1

    MOVE.W   D0, D2           ;kopiujemy pozycję X do D2
    LSR.W    #3, D2          ;który bajt w linii dla pozycji X?
    MULU     #40, D1         ;która linia?
    ADD.W    D2, D1          ;który bajt pamięci?
    NOT      D0              ;określenie bitu w bajcie.
    BSET     D0, (A1, D1.w)   ;postawienie punktu.

    DBF      D7, Set_Loop
    RTS
```

Rozkaz NOT (Logical Complement — dopełnienie logiczne)

Składnia:

NOT <ea>

Atrybuty:

rozmiar: B, W, L

Działanie:

Instrukcja NOT umożliwia otrzymanie zanegowanego operandu przeznaczenia. Wszystkie zera operandu są zamieniane na jedynki, a jedynki na zera. Rozmiar operandu jest dowolny. Instrukcję tę można zastąpić przez EOR.B #\$ff, EOR.W #\$ffff lub EOR.L #\$ffffffff.

Kody warunków:

- X — nie zmieniany,
 N — ustawiany, gdy wynik jest ujemny, w przeciwnym wypadku zerowany.
 Z — ustawiany, gdy wynik wynosi 0, w przeciwnym wypadku zerowany.
 V — zawsze zerowany,
 C — zawsze zerowany.

Dozwolone tryby adresowania:

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
tak	nie	tak	tak	tak	tak	tak
X.w	X.1	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	nie	nie	nie	nie	nie

Przykład:

```
MOVE.B  #011010011,D0
NOT.B   D0
```

Po wykonaniu tych dwóch instrukcji, w rejestrze D0 zostanie umieszczona wartość %00101100.

10.5.2 Reprezentacja szkieletowa

Metoda szkieletowa polega na zdefiniowaniu struktury, opisującej krawędzie obiektu. Niech pierwsze słowo określa ilość krawędzi minus 1, a kolejne dwa — numery współrzędnych początku i końca krawędzi w tablicy **2D_Coords**.

```
Cube_Struct_Lines:
    DC.W    12-1                ; ilość linii-1
    DC.W    0,1                 ; od P1 do P2
    DC.W    1,2                 ; od P2 do P3
    DC.W    2,3                 ; od P3 do P4
    DC.W    3,0                 ; od P4 do P1
    DC.W    4,5                 ; od P5 do P6
    DC.W    5,6                 ; od P6 do P7
    DC.W    6,7                 ; od P7 do P8
    DC.W    7,4                 ; od P8 do P5
    DC.W    1,4                 ; od P2 do P5
    DC.W    2,5                 ; od P3 do P6
    DC.W    3,6                 ; od P4 do P7
    DC.W    4,7                 ; od P5 do P8
```

Jeśli mamy już stworzoną tablicę połączeń odcinków, to jak ją możemy wykorzystać? Oto przykład:

```
Rysuj_Linie:
    LEA     2D_Coords,A0
    LEA     Cube_Struct_Lines,A1
    MOVE.W  (A1)+,D7
```

```

Petla_Rysowania:
    MOVE.W (A1)+,D3      ;początkowa współrzędna krawędzi
    MOVE.W (A1)+,D4      ;końcowa współrzędna krawędzi
    ASL.W #2,D3           ;Mnożymy przez 4. Wynik wskazuje przesunięcie
    ASL.W #2,D4           ;względem początku tablicy, dla poszukiwanej
                          ;pary współrzędnych
    MOVE.W (A0,D3.w),D0   ;początkowa współrzędna X
    MOVE.W 2(A0,D3.w),D1  ;początkowa współrzędna Y
    MOVE.W (A0,D4.w),D2   ;końcowa współrzędna X
    MOVE.W 2(A0,D4.w),D3  ;końcowa współrzędna Y

    BSR     Draw_Line     ;narysuj linię

    DBF     D7,Petla_Rysowania
    RTS

```

Pewnie zastanawiasz się, czemu mnożymy przez 4? Otóż para współrzędnych X i Y zajmuje 4 bajty. My mamy numer tylko numer pary współrzędnych. Mnożąc przez 4, otrzymujemy przesunięcie względem początku tablicy wierzchołków, które wskazuje nam położenie współrzędnych w pamięci.

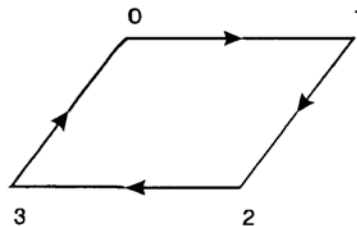
10.5.3 Reprezentacja szkieletowa z usuniętymi niewidocznymi liniami

Zaprezentowana powyżej metoda szkieletowa często jest niewystarczająca. Jeśli chcemy, żeby nasza kostka była bardziej realistyczna to musimy usunąć te linie, które powinny być niewidoczne. Aby tego dokonać, musimy nieco zmodyfikować nasz program.

Wyznaczamy wszystkie wierzchołki i zamiast odcinków, łączymy je w ściany. Opisujemy je podobnie jak krawędzie, ale:

- każda ściana jest opisana ciągiem wierzchołków, zgodnie z ruchem wskazówek zegara (wszystkie odcinki będą rysowane w jednym kierunku),
- opis każdej ściany zakończony jest liczbą -1,
- koniec struktury ściany oznaczamy liczbą \$8000.

Oto przykładowy rysunek.



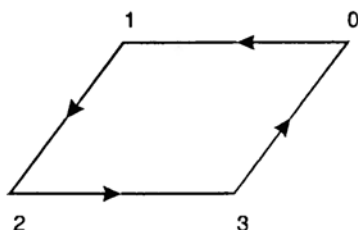
Opis ściany przedstawionej na powyższym rysunku będzie wyglądał następująco:

```

DC.W 0,1,2,3,0,-1
DC.W $8000

```

Skoro układ wszystkich wierzchołków tej ściany jest prawoskrętny, to możemy tą cechę wykorzystać do sprawdzania widoczności ściany. Zwróćmy uwagę, co się stanie gdy obrócimy ścianę wokół osi Y o kąt większy niż 90°.



Jak widać, układ ściany (jej kierunek rysowania) zmienił się z prawoskrętnego na lewoskrętny, a ściana jest obrócona do nas „tyłem”.

Jeśli teraz zdefiniujemy wszystkie ściany, to wystarczy za każdym razem po obrotach sprawdzać, które ściany są do nas obrócone „tyłem”, czyli są niewidoczne. Aby sprawdzić, czy ściana jest widoczna, należy obliczyć następujące wyrażenie:

$$w = (x - x_1) \cdot (y_3 - y_2) - (x_2 - x_3) \cdot (y_2 - y_1)$$

Jeśli wynik tego wyrażenia będzie ujemny, ściana obrócona jest do nas tyłem i jest dla nas niewidoczna.

Jako $(x_1; y_1)$, $(x_2; y_2)$ i $(x_3; y_3)$ przyjmujemy 3 kolejne wierzchołki badanej ściany obliczone ze wzoru na perspektywę.

Procedura testująca widoczność ścian i rysująca tylko widoczne wyglądać będzie tak:

```

...
...
...
BSR.W   Testuj_Widoczność
BSR.W   Rysuj_Ściany
...
...
...
;-----
Testuj_Widoczność:
    LEA    VPol, A6
    LEA    2D_Coords, A5
    LEA    List2D, A4
Testuj:
    CMP.W  #$8000, (a6)
    BEQ.B  Koniec_Testowania

    MOVEM.W (A6), D0-D2

    LSL.W  #2, D0
    LSL.W  #2, D1
    LSL.W  #2, D2

    MOVEM.W (A5, D0.w), D0/D3
    MOVEM.W (A5, D1.w), D1/D4
    MOVEM.W (A5, D2.w), D2/D5

    SUB.W  D1, D2
    SUB.W  D0, D1
    SUB.W  D4, D5
    SUB.W  D3, D4

```

```

        Muls      D1,D5
        Muls      D4,D2
        sub.l      D2,D5
        bmi.b      Widoczna
        bra.b      Niewidoczna
Ściana_Widoczna:
        move.l     A6,(A4)+
Niewidoczna:
        cmp.w      #-1,(A6)+
        bne.b      Niewidoczna
        bra.b      Testuj
Koniec_Testowania:
        move.l     #0,(A4)+
        RTS

;-----
Rysuj_Ściany:
        lea        List2D,A6
Pętla1:
        move.l     (A6)+,A2
        cmp.l      #0,A2
        beq        Koniec_Rysowania
Pętla2:
        move.w     (A2)+,D3
        move.w     (A2),D4
        cmp.w      #-1,D4
        beq        Pętla1

        asl.w      #2,D3
        asl.w      #2,D4

        move.w     (A0,D3.w),D0
        move.w     2(A0,D3.w),D1
        move.w     (A0,D4.w),D2
        move.w     2(A0,D4.w),D3

        bsr        Draw_Line

        bra        Pętla2

Koniec_Rysowania:
        RTS

Vpol:
        dc.w       0,1,2,3,0,-1
        dc.w       $8000,$8000

List2D:
        blk.b      500,0
Drawtable:
        blk.b      1000,0

```

10.5.4 Reprezentacja grafiki wypełnianej

Rysowanie grafiki wypełnianej, to już nieco większy kłopot. Musimy przecież określić kolor ściany, i odpowiednio ją wypełniać.

Grafikę wypełnianą możemy podzielić na:

- „wypukłą” np. kostka, gdzie nie może się zdarzyć, że jedna ściana zasłania fragment ściany drugiej,
- „niewypukłą” np. statek kosmiczny, gdzie może się zdarzyć, że jedna ściana zasłania fragment ściany drugiej.

10.5.4.1 Wektorówka „wypukła”

Korzystając z założenia, że jedna ściana nie może nachodzić na drugą, znacznie upraszczamy sytuację. Wystarczy tylko stworzyć odpowiednią strukturę ścian, procedurę wypełniania blitterem, a także nieco zmodyfikować procedurę rysowania linii.

Dla tych, którzy jeszcze nie potrafili do końca zebrać informacji o grafice wektorowej i złożyć z nich procedur animacji, w rozdziale 10.6 przedstawiam kompletną procedurę grafiki wypełnianej, „wypukłej”.

10.5.4.2 Wektorówka „niewypukła”

Zastanówmy się teraz, dlaczego poprzednia procedura grafiki przestrzennej nie mogła sobie poradzić z bardziej skomplikowanymi bryłami, na przykład dwoma kostkami? Działo się tak dlatego, ponieważ przy sprawdzaniu widoczności ścian mogły zajść tylko dwie sytuacje: ściana widoczna lub niewidoczna. Tak więc, jeśli dwie ściany nachodziły na siebie, to na ekranie wyglądało to tak:



a powinno wyglądać:



Jeśli chcemy teraz przedstawiać na ekranie figury niewypukłe, to niestety, musimy zmodyfikować nieco naszą procedurę.

W wypadku, gdy istnieje możliwość zasłaniania fragmentu ściany przez inną, musimy zrezygnować z poprzedniej metody rysowania i wymyślić nową. Najskuteczniej możemy to zrobić poprzez oddzielne rysowanie każdej ze ścian i „narzucanie” jej na ekran. Wcześniej jednak musimy

określić, które ze ścian znajdują się bliżej, a które dalej od obserwatora. Jeśli tego nie zrobimy, to może okazać się, że ściany najbardziej oddalone będą rysowane jako ostatnie i zasłonią te, które faktycznie są najbliżej.

Omówię teraz najczęściej używane metody rysowania obiektów niewypukłych.

a. Metoda powierzchniowa

Metoda ta polega na użyciu dodatkowej pamięci na **obszar roboczy**. Rysujemy na nim kolejne ściany, wypełniamy je i „narzucamy” na ekran.

Żeby uchronić się przed kopiowaniem całego obszaru roboczego, wyznaczamy najmniejsze i największe wartości X i Y ściany. Wtedy:

$$\text{wysokość blitu} = y_{\max} - y_{\min}$$

$$\text{szerokość blitu} = x_{\max} - x_{\min}$$

Na podstawie tych wartości obliczamy modulo i możemy zacząć się bawić.

Oto algorytm:

- obierz adres struktury ściany,
- wyznacz maksymalne i minimalne współrzędne x i y,
- narysuj całą ścianę w obszarze roboczym,
- oblicz modulo, szerokość i wysokość blitu,
- wypełnij prostokąt zawierający ścianę (określany przez $(x_{\min}; y_{\min})$ i $(x_{\max}; y_{\max})$),
- „narzuc” ścianę na odpowiednie bitplany ekranu, w celu uzyskania koloru,
- wyczyść prostokąt ściany w obszarze roboczym,
- obierz adres struktury kolejnej ściany,
- idź do punktu 2.

Jak widać, metoda ta jest dość powolna w porównaniu z wektorówką wypukłą. Powodem tego jest bardzo duża liczba blitów do wykonania, przez co blitter nie może nadążyć z kopiowaniem takiej ilości danych, a główny procesor czeka beczynnie na zakończenie danego blitu.

W celu określenia, które ściany znajdują się bliżej obserwatora, musimy je posortować (ustawić w kolejności) według jakiegoś wyznacznika. Najprościej można to zrobić obliczając średnią współrzędną Z ściany.

Pozostaje jeszcze jedna sprawa: jak ściany przenosić na ekran? Jeśli narysowaliśmy w buforze (naszym obszarze roboczym) ścianę, to musimy wiedzieć w jakim kolorze ma być rysowana. Przypuśćmy, że kolor ściany określa rejestr COLOR06. Liczba 6 wskazuje, że ściana ma być narysowana na bitplanach 2 i 3, a bitplan 1 jest pusty. Wynika to z przedstawienia binarnego liczby $6 = \%110$. Tak więc przy przenoszeniu ściany na ekran, musimy użyć operacji OR z bitplanami 2 i 3, oraz wyczyścić bitplan 1 przez operację AND z buforem jako maską.

b. Metoda składania z obiektów wypukłych

Metoda składania obiektów niewypukłych z wypukłych jest nieco szybsza, chociaż mniej uniwersalna. Jej idea jest zbliżona do metody powierzchniowej, ale zamiast ścian, w buforze, rysujemy całe obiekty wypukłe, które potem przenosimy na ekran. Musimy pamiętać jednak o następujących rzeczach:

- bufor musi mieć tyle samo bitplanów co ekran,
- rysujemy tylko widoczne ściany danego obiektu wypukłego,
- sortujemy obiekty według średniej Z-ów wszystkich wierzchołków.

10.6 Kompletna procedura prezentacji wypełnianej grafiki przestrzennej

```

-----
Xmin      =      0      ;minimalna współrzędna X
Ymin      =      0      ;minimalna współrzędna Y
Xmax      =      319    ;maksymalna współrzędna X
Ymax      =      255    ;maksymalna współrzędna Y
Mod8      =      40     ;szerokość ekranu w bajtach
Ilp       =      2      ;ilość bitplanów

;-----
; Definiujemy procedury typu "makro"
Blit:      Macro
.\@1      BTST      #14,$dff002
          BNE       .\@1
          Endm

;-----
Wait_Beam: Macro
.\@1      MOVE.L    $dff004,d0
          LSR.L     #8,D0
          ANDI.W    #$1ff,d0
          CMPI.W    #$130,d0
          BNE       .\@1
          Endm

;-----
Section Code,Code_P

Start:
          MOVEM.L   D0-A6,-(A7)      ;rejestry na stos
          MOVE.L    #Copper,$dff080 ;uruchamiamy naszą copperlistę
          MOVE.W    #$4000,$dff09a  ;wyłączamy przerwania

Main_Loop:
          Blit
          Wait_Beam
          BSR       Czyszc_Ekran_Roboczy ;czyszczenie ekranu roboczego
          BSR       Przeliczanie         ;przeliczanie z 3D na 2D
          BSR       Testuj_Widocznosc    ;testowanie widoczności ścian
          BSR       Rysuj_Sciany         ;rysowanie ścian na ekranie
          BSR       Fill_Screen          ;wypełnienie całego ekranu
          BSR       Zamien_Bufory        ;zamiana buforów (ekranów)
          BTST      #6,$bfe001
          BNE.B     Main_Loop
          MOVE.W    #$c000,$dff09a      ;włączamy przerwania
          MOVE.L    4.w,A6
          MOVE.L    156(A6),A1

```

```

MOVE.L 38(A1), $dff080 ;uruchamiamy systemową copperlistę
MOVE.W #0, $dff088
MOVE.W #$8020, $dff096 ;włączamy z powrotem DMA spriteów
MOVEM.L (A7)+, D0-A6
MOVEQ #0, D0 ;informujemy system, że nie było
RTS ;żadnego błędu

;-----
Przeliczanie: MOVEM.L D0-A6, -(Sp)

ADDQ.W #2, Anglex
ADDQ.W #2, Angley
ADDQ.W #2, Anglez

AND.W #$1ff, anglex
AND.W #$1ff, angley
AND.W #$1ff, anglez

MOVE.W Anglex, A0
MOVE.W Angley, A1
MOVE.W Anglez, A2

LEA Cube_Coords, A3
LEA Coords_2D, A4

MOVE.W (A3)+, D7

Petla:
MOVEM.W (A3)+, D0-D2
BSR.W Rotates
ADD.W X_Add, D0
ADD.W Y_Add, D1
ADD.W Z_Add, D2
EXT.L D0
EXT.L D1
ASL.L #8, D0
ASL.L #8, D1
DIVS D2, D0
DIVS D2, D1
ADDI.W #(Xmax+1)/2-1, D0
ADDI.W #(Ymax+1)/2-1, D1
MOVE.W D0, (A4)+
MOVE.W D1, (A4)+
DBF D7, Petla
MOVEM.L (Sp)+, D0-A6
RTS

;----- Testowanie widoczności ścian

Testuj_Widocznosc:
MOVEM.L D0-A6, -(Sp)
LEA Cube_Faces, A6
LEA Coords_2D, A5
LEA List2D, A4

Testuj_Dalej:
CMP.W #$8000, (a6)
BEQ.S Koniec_Testowania

MOVEM.W 2(A6), D0-D2

LSL.W #2, D0
LSL.W #2, D1
LSL.W #2, D2

MOVEM.W (A5, D0.w), D0/D3 ;pobieramy pierwsze trzy wierzchołki
;ściany
MOVEM.W (A5, D1.w), D1/D4
MOVEM.W (A5, D2.w), D2/D5

```



```

;rozkaz MOVEM automatycznie rozszerza
;zawartość rejestrów do długiego słowa
;dlatego bez obaw możemy odejmować
;długie słowa
SUB.L D1,D2
SUB.L D0,D1
SUB.L D4,D5
SUB.L D3,D4

MULS D1,D5
MULS D4,D2
SUB.L D2,D5
BGE Widoczna
BRA.S Niewidoczna
Widoczna:
MOVE.L A6,(A4)+
Niewidoczna:
CMP.W #-1,(A6)+
BNE.B Niewidoczna
BRA.B Testuj_Dalej
Koniec_Testowania:
MOVE.L #0,(A4)+
MOVEM.L (Sp)+,D0-A6
RTS

;-----
Zamien_Bufory: MOVEM.L D0-A6,-(Sp)
MOVE.L Show_Plane,A0 ;zamieniamy ekrany
MOVE.L Work_Plane,Show_Plane
MOVE.L A0,Work_Plane
MOVEQ #Ilp-1,D7
LEA Plane_Adrs,A0
MOVE.L Show_Plane,D0
Set_Planes: SWAP D0
MOVE.W D0,2(A0)
SWAP D0
MOVE.W D0,6(A0)
ADD.L #8,A0
ADD.L #(Ymax+1)*Mod8,D0
DBF D7,Set_Planes
MOVEM.L (Sp)+,D0-A6
RTS

Czyszc_Ekran_Roboczy:
MOVEM.L A6,-(Sp)
Blit
LEA $dff000,a6
MOVE.L Work_Plane,$54(a6)
MOVE.W #0,$66(a6)
MOVE.W #$0000000100000000,$40(a6)
MOVE.W #$0000,$42(a6)
MOVE.W #0,$66(a6)
MOVE.W #(Ymax+1)*Ilp*64+(Mod8/2),$58(a6)
MOVEM.L (Sp)+,A6
RTS

Rotates:
;-----
;Wejście: D0-X, D1-Y, D2-Z
;Wyjście: D0-X', D1-Y', D2-Z'
Obroty:
MOVEM.L D3-D7/A3-A6,-(SP)

LEA Cos,A3
LEA Sin,A4

BSR.S OS_X
BSR.S OS_Y
BSR.S OS_Z

MOVEM.L (SP)+,D3-D7/A3-A6
RTS

```

```

; X'=X
; Y'=Y*cos(alfa)-Z*sin(alfa)
; Z'=Y*sin(alfa)+Z*cos(alfa)
OS_X:
    MOVE.W D2,D5
    MOVE.W D1,D4
    MULS    (A4,A0.W),D5    ;Z*sin(alfa)
    MULS    (A3,A0.W),D4    ;Y*cos(alfa)
    SUB.L   D5,D4           ;Y*cos(alfa)-Z*sin(alfa)
    ASR.L   #8,D4           ;dzielimy przez 256, poniewaz kazdy element
                           ;tablicy byl pomnozony przez tyle samo

    MOVE.W D4,D6

    MOVE.W D2,D4
    MOVE.W D1,D5
    MULS    (A4,A0.W),D5    ;Y*sin(alfa)
    MULS    (A3,A0.W),D4    ;Z*cos(alfa)
    ADD.L   D4,D5           ;Y*sin(alfa)+Z*cos(alfa)
    ASR.L   #8,D5

    MOVE.W D6,D2
    MOVE.W D5,D1
    RTS

; X'=X*cos(beta)-Z*sin(beta)
; Y'=Y
; Z'=X*sin(beta)+Z*cos(beta)
OS_Y:
    MOVE.W D2,D5
    MOVE.W D0,D4
    MULS    (A4,A1.W),D5    ;Z*sin(beta)
    MULS    (A3,A1.W),D4    ;X*cos(beta)
    SUB.L   D5,D4           ;X*cos(beta)-Z*sin(beta)
    ASR.L   #8,D4

    MOVE.W D4,D6

    MOVE.W D2,D4
    MOVE.W D0,D5
    MULS    (A4,A1.W),D5    ;X*sin(beta)
    MULS    (A3,A1.W),D4    ;Z*cos(beta)
    ADD.L   D4,D5           ;X*sin(beta)+Z*cos(beta)
    ASR.L   #8,D5

    MOVE.W D6,D0
    MOVE.W D5,D2
    RTS

; X'=X*cos(gamma)-Y*sin(gamma)
; Y'=X*sin(gamma)+Y*cos(gamma)
; Z'=Z
OS_Z:
    MOVE.W D1,D5
    MOVE.W D0,D4
    MULS    (A4,A2.W),D5    ;Y*sin(gamma)
    MULS    (A3,A2.W),D4    ;X*cos(gamma)
    SUB.L   D5,D4           ;X*cos(gamma)-Y*sin(gamma)
    ASR.L   #8,D4

    MOVE.W D4,D6

    MOVE.W D1,D4
    MOVE.W D0,D5
    MULS    (A4,A2.W),D5    ;X*sin(gamma)
    MULS    (A3,A2.W),D4    ;Y*cos(gamma)
    ADD.L   D4,D5           ;X*sin(gamma)+Y*cos(gamma)
    ASR.L   #8,D5

    MOVE.W D6,D0
    MOVE.W D5,D1
    RTS

```

```

;-----
Rysuj_Sciany:      LEA      Coords_2D,A0
                   LEA      List2D,A6

Petla1:            MOVE.L   (A6)+,A2
                   CMP.L    #0,A2
                   BEQ       Koniec_Rysowania
                   MOVE.W    (A2)+,Color      ; w COLOR zapamiętujemy kolor ściany

Petla2:            MOVE.W    (A2)+,D3
                   MOVE.W    (A2),D4
                   CMP.W     #-1,D4
                   BEQ        Petla1

                   LSL.W     #2,D3
                   LSL.W     #2,D4

                   MOVE.W    (A0,D3.w),D0
                   MOVE.W    2(A0,D3.w),D1
                   MOVE.W    (A0,D4.w),D2
                   MOVE.W    2(A0,D4.w),D3

                   JSR        Draw

                   BRA        Petla2

Koniec_Rysowania:  RTS

;-----
Fill_Screen:       Blit
                   LEA        $dff000,a6
                   MOVE.L     #$09f0001a,$40(a6)
                   MOVE.L     #$ffffffff,$44(a6)
                   MOVE.L     Work_Plane,D0
                   ADD.L      #1lp*(Ymax+1)*((Xmax+1)/8)-1,D0
                   MOVE.L     D0,$50(a6)
                   MOVE.L     D0,$54(a6)
                   MOVE.W     #0,$64(a6)
                   MOVE.W     #0,$66(a6)
                   MOVE.W     #(Ymax+1)*1lp*64+((Xmax+1)/16),$58(a6)
                   RTS

;-----
Draw:              MOVEM.L    D0-A6,-(SP)      ; rysujemy linie
                   LEA        $dff000,a6
                   CMP.W      D1,D3
                   BEQ.W      No_Line
                   BGE.B      P_1
                   EXG         D2,D0
                   EXG         D1,D3

P_1:               ADDQ.B      #1,D1
                   MOVEQ       #$f,d4
                   AND.B        D2,D4
                   ROR.L        #$4,d4
                   SUB.W        D3,D1
                   NEG.W         D1

                   MULS         #Mod8,D3

                   SUB.W        D2,D0
                   BLT.B        P_5
                   CMP.W        D0,D1
                   BGE.B        P_4
                   ORI.L        #$0b4a001b,d4
                   BRA.B         P_9

P_4:               ORI.L        #$0b4a0005,d4
                   EXG          D0,D1
                   BRA.B         P_9

```

```

P_5:      NEG.W    D0
          CMP.W    D0,D1
          BGE.B    P_8
          ORI.L    #0b4a001f,d4
          BRA.B    P_9

P_8:      ORI.L    #0b4a000f,d4
          EXG      D0,D1

P_9:      ASL.W    #1,D1
          ASR.W    #3,D2
          ANDI.L   #0000ffff,d2
          ADD.L    D2,D3
          MOVE.W   D1,D2
          SUB.W    D0,D2
          SUB.W    D0,D2
          BGE.B    P_10
          BSET     #6,D4

P_10:     ASL.W    #6,d0
          ADD.W    #42,d0

          Blit
          MOVE.W   D1,$62(a6)
          MOVE.W   D2,$64(a6)
          MOVE.L   #-1,$44(a6)
          MOVE.W   #Mod8,$60(a6)
          MOVE.L   #ffff8000,$72(a6)
          ADD.L    Work_Plane,D3

          MOVE.W   Color,D1
          MOVE.W   #ilp-1,D6      ;rysujemy linię na odpowiednich
Draw_Loop:                                     ;bitplaneach (zgodnie z wartością
          BTST     #0,D1          ;w Color) w celu uzyskania koloru
          BEQ.B    Loopn
          MOVE.W   D2,$52(a6)
          MOVE.L   D4,$40(a6)
          MOVE.L   D3,$48(a6)
          MOVE.L   D3,$54(a6)
          MOVE.W   D0,$58(a6)

Loopn:    Blit
          LSR.W    #1,D1
          ADDI.L   #(Ymax+1)*(Xmax+1)/8,D3
          DBF      D6,Draw_Loop

No_Line:  MOVEM.L  (SP)+,D0-A6
          RTS

;-----

Section Copper,Data_C

Copper:   DC.L     $00960020
          DC.L     $008e2c81
          DC.L     $00902cc1
          DC.L     $00920038,$009400d0
          DC.L     $01080000,$010a0000
          DC.L     $01020000,$01040000
          DC.L     $01800000
          DC.L     $01820666
          DC.L     $01840888
          DC.L     $01860aaa
          DC.L     $01000200

Plane_Adrs: DC.L    $00e00000,$00e20000
            DC.L    $00e40000,$00e60000
            DC.W    $0100,$1000*ilp+$200
            DC.L    $fffffffe

```

Section Datas,Data_P

Cube_Coords:

DC.W	7		
DC.W	-800,	800,	-800
DC.W	800,	800,	-800
DC.W	800,	-800,	-800
DC.W	-800,	-800,	-800
DC.W	-800,	800,	800
DC.W	800,	800,	800
DC.W	800,	-800,	800
DC.W	-800,	-800,	800

Cube_Faces:

DC.W	1,0,1,2,3,0,-1
DC.W	1,7,6,5,4,7,-1
DC.W	2,1,5,6,2,1,-1
DC.W	2,4,0,3,7,4,-1
DC.W	3,1,0,4,5,1,-1
DC.W	3,2,6,7,3,2,-1
DC.W	\$8000,\$8000

Sin:

DC.W	1,-5,-12,-18,-24,-30,-37,-43
DC.W	-49,-55,-61,-67,-73,-79,-85,-91
DC.W	-97,-103,-108,-114,-120,-125,-131,-136
DC.W	-141,-146,-151,-156,-161,-166,-171,-176
DC.W	-180,-184,-189,-193,-197,-201,-205,-208
DC.W	-212,-215,-219,-222,-225,-228,-230,-233
DC.W	-236,-238,-240,-242,-244,-246,-247,-249
DC.W	-250,-251,-252,-253,-254,-254,-255,-255
DC.W	-255,-255,-255,-254,-254,-253,-252,-251
DC.W	-250,-249,-247,-246,-244,-242,-240,-238
DC.W	-236,-233,-230,-228,-225,-222,-219,-215
DC.W	-212,-208,-205,-201,-197,-193,-189,-184
DC.W	-180,-176,-171,-166,-161,-156,-151,-146
DC.W	-141,-136,-131,-125,-120,-114,-108,-103
DC.W	-97,-91,-85,-79,-73,-67,-61,-55
DC.W	-49,-43,-37,-30,-24,-18,-12,-5
DC.W	1,6,13,19,25,31,38,44
DC.W	50,56,62,68,74,80,86,92
DC.W	98,104,109,115,121,126,132,137
DC.W	142,147,152,158,162,167,172,177
DC.W	181,185,190,194,198,202,206,209
DC.W	213,216,220,223,226,229,231,234
DC.W	237,239,241,243,245,247,248,250
DC.W	251,252,253,254,255,255,256,256
DC.W	256,256,256,255,255,254,253,252
DC.W	251,250,248,247,245,243,241,239
DC.W	237,234,231,229,226,223,220,216
DC.W	213,209,206,202,198,194,190,185
DC.W	181,177,172,167,162,157,152,147
DC.W	142,137,132,126,121,115,109,104
DC.W	98,92,86,80,74,68,62,56
DC.W	50,44,38,31,25,19,13,6

Cos:

DC.W	256,256,256,255,255,254,253,252
DC.W	251,250,248,247,245,243,241,239
DC.W	237,234,231,229,226,223,220,216
DC.W	213,209,206,202,198,194,190,185
DC.W	181,177,172,167,162,157,152,147
DC.W	142,137,132,126,121,115,109,104
DC.W	98,92,86,80,74,68,62,56
DC.W	50,44,38,31,25,19,13,6
DC.W	1,-5,-12,-18,-24,-30,-37,-43
DC.W	-49,-55,-61,-67,-73,-79,-85,-91
DC.W	-97,-103,-108,-114,-120,-125,-131,-136
DC.W	-141,-146,-151,-156,-161,-166,-171,-176
DC.W	-180,-184,-189,-193,-197,-201,-205,-208
DC.W	-212,-215,-219,-222,-225,-228,-230,-233
DC.W	-236,-238,-240,-242,-244,-246,-247,-249

```

DC.W      -250,-251,-252,-253,-254,-254,-255,-255
DC.W      -255,-255,-255,-254,-254,-253,-252,-251
DC.W      -250,-249,-247,-246,-244,-242,-240,-238
DC.W      -236,-233,-230,-228,-225,-222,-219,-215
DC.W      -212,-208,-205,-201,-197,-193,-189,-184
DC.W      -180,-176,-171,-166,-161,-156,-151,-146
DC.W      -141,-136,-131,-125,-120,-114,-108,-103
DC.W      -97,-91,-85,-79,-73,-67,-61,-55
DC.W      -49,-43,-37,-30,-24,-18,-12,-5
DC.W      1,6,13,19,25,31,38,44
DC.W      50,56,62,68,74,80,86,92
DC.W      98,104,109,115,121,126,132,137
DC.W      142,147,152,158,162,167,172,177
DC.W      181,185,190,194,198,202,206,209
DC.W      213,216,220,223,226,229,231,234
DC.W      237,239,241,243,245,247,248,250
DC.W      251,252,253,254,255,255,256,256

;-----
Old_Copper:   DC.L      0
Work_Plane:   DC.L      Screen1
Show_Plane:   DC.L      Screen2
Anglez:       DC.W      0
Anglex:       DC.W      0
Angley:       DC.W      0
Color:        DC.W      0
X_Add:        DC.W      0
Y_Add:        DC.W      0
Z_Add:        DC.W      4000
;-----
                Section Screens,Bss_C
Screen1:       Ds.b      (Ymax+1)*Mod8*1lp
Screen2:       Ds.b      (Ymax+1)*Mod8*1lp
;-----
                Section Tables,Bss_C
List2D:       Ds.b      1024
Coords_2D:    Ds.b      1024
;-----

```

Rozkaz EXG (Exchange registers — zamiana rejestrów)

Składnia:

EXG Rx,Ry

Atrybuty:

rozmiar: L

Działanie:

Instrukcja EXG wymienia ze sobą zawartości dwóch rejestrów. Jest to zawsze operacja o rozmiarze długiego słowa. Instrukcja ta pracuje w trzech trybach:

- zamiana rejestrów danych,
- zamiana rejestrów adresowych,
- zamiana rejestru danych z rejestrem adresowym.

Kody warunków:

X — nie zmieniany,

N — nie zmieniany,

Z — nie zmieniany,

V — nie zmieniany,

C — nie zmieniany.

Dozwolone tryby adresowania:

tylko bezpośredniego adresowania rejestrów.

Przykład:

EXG D0, A0

Zawartość rejestrów D0 i A0 zostanie ze sobą zamieniona.

Rozkaz CMPI (Compare Immediate — porównanie natychmiastowe)

CMPI #<dana>, <ea>

Atrybuty:

rozmiar: B, W, L

Działanie:

Instrukcja CMPI porównuje daną natychmiastową z operandem określonym adresem efektywnym. Po odjęciu danej natychmiastowej od operandu przeznaczenia, odpowiednio ustawiane są kody warunków. Żaden z operandów nie zostaje zmieniony. Rozmiar instrukcji może być określony jako bajt, słowo lub długie słowo. Rozmiar danej natychmiastowej musi być zgodny z rozmiarem instrukcji.

Kody warunków:

X — nie zmieniany,

N — ustawiany gdy wynik jest ujemny, w przeciwnym wypadku zerowany,

Z — ustawiany gdy wynik jest zerowy, w przeciwnym wypadku zerowany,

V — ustawiany gdy wystąpił nadmiar przy odejmowaniu, w przeciwnym wypadku zerowany,

C — ustawiany gdy wygenerowana została pożyczka, w przeciwnym wypadku zerowany.

Dozwolone tryby adresowania:

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
tak	nie	tak	tak	tak	tak	tak
X.w	X.1	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	nie	nie	nie	nie	nie

Przykład:

```

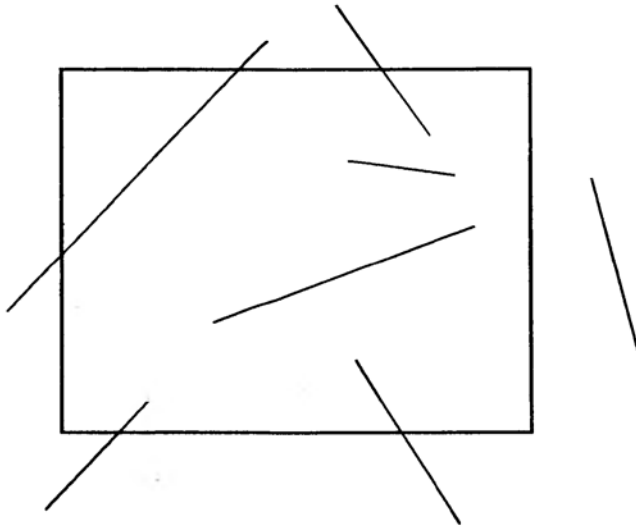
Loop:  MOVEQ    #0, D0
        ADD.W   #1, D0
        CMPI.W  #20, D0
        BNE     Loop

```

Pętla będzie wykonywana do momentu, w którym zawartość rejestru D0 wyniesie 20.

10.7 Clipping

Pewnie zdążyłeś już zauważyć, że procedura rysowania linii ma jedną podstawową wadę. Przy podaniu większych współrzędnych (na przykład (400;600)), linia jest rysowana gdzieś w pamięci, co może doprowadzić do zawieszenia się komputera. Problem rozwiązałoby obcinanie odcinka — kreślenie tylko tego fragmentu, który znajduje się na ekranie.



Obcinanie polega na wyznaczeniu tych współrzędnych należących do odcinka, które nie przekraczają danych wartości. Wyznamy więc współrzędne punktów ograniczające nasz prostokątny ekran:

$$x_{\min} = 0$$

$$y_{\min} = 0$$

$$x_{\max} = 319$$

$$y_{\max} = 255$$

Niech $(x1;y1)$ oraz $(x2;y2)$ będą współrzędnymi początku i końca odcinka.

Punkty przecięcia z krawędziami ekranu można wyliczyć na podstawie następujących wzorów:

- lewa krawędź:

$$y1' = y1 + (x_{\min} - x1) \cdot \frac{y2 - y1}{x2 - x1}$$

$$x1' = x_{\min}$$

- prawa krawędź:

$$y1' = y1 + (x_{\max} - x1) \cdot \frac{y2 - y1}{x2 - x1}$$

$$x1' = x_{\max}$$

- górna krawędź:

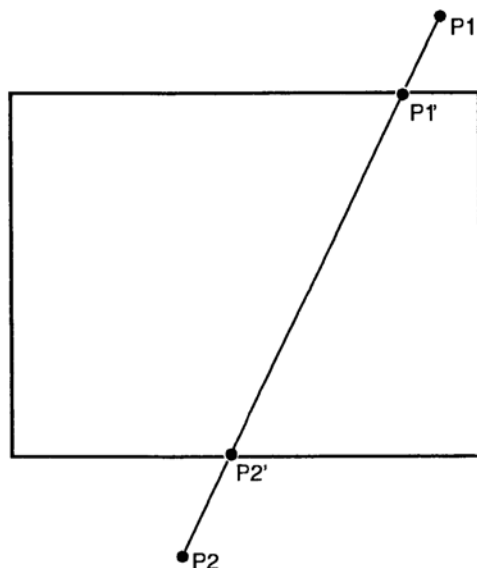
$$y1' = y_{\min}$$

$$x1' = x1 + (y_{\min} - y1) \cdot \frac{x2 - x1}{y2 - y1}$$

- dolna krawędź:

$$y1' = y_{\max}$$

$$x1' = x1 + (y_{\max} - y1) \cdot \frac{x2 - x1}{y2 - y1}$$



A tak będzie wyglądać cała procedura obcinania odcinka:

```

;-----
;Wejście:
;      D0-X1 ; D1-Y1
;      D2-X2 ; D3-Y2
;-----
XMIN=0
XMAX=319
YMIN=0
YMAX=255

```

```

Clipping:      CMP.W   D1,D3
               BGE.B   Ay2w
               EXG     D0,D2
               EXG     D1,D3
Ay2w:          CMPI.W  #YMAX,D1
               BGT.W   Aend
               CMPI.W  #YMIN,D3
               BLT.W   Aend
               CMPI.W  #YMAX,D3
               BLE.B   Any
               SUB.W   D0,D2
               MOVE.W  #YMAX,D4
               SUB.W   D1,D4
               MULS.W  D4,D2
               SUB.W   D1,D3
               BEQ.B   Anie1
               DIVS.W  D3,D2
Anie1:         ADD.W   D0,D2
               MOVE.W  #YMAX,D3
Any:           CMP.W   #YMIN,D1
               BGE.B   Ax1
               SUB.W   D2,D0
               MOVE.W  #YMIN,D4
               SUB.W   D3,D4
               MULS.W  D4,D0
               SUB.W   D3,D1
               BEQ.B   Anie2
               DIVS.W  D1,D0
Anie2:         ADD.W   D2,D0
               MOVE.W  #YMIN,D1
Ax1:           CMP.W   D0,D2
               BGE.B   Ax2w
               EXG     D0,D2
               EXG     D1,D3
Ax2w:          CMP.W   #XMIN,D2
               BLT.B   Aend
               CMPI.W  #XMAX,D0
               BLE.B   Anx
               MOVE.W  #XMAX,D0
               MOVE.W  D0,D2
Anx:           CMP.W   #XMIN,D0
               BGE.B   Ay
               SUB.W   D3,D1
               MOVE.W  #XMIN,D4
               SUB.W   D2,D4
               MULS.W  D4,D1
               SUB.W   D2,D0
               BEQ.B   Anie3
               DIVS.W  D0,D1
Anie3:         ADD.W   D3,D1
               MOVE.W  #XMIN,D0
Ay:            CMPI.W  #XMAX,D2
               BLE.W   Ary
               SUB.W   D1,D3
               MOVE.W  #XMAX,D4
               SUB.W   D0,D4
               MULS.W  D4,D3
               SUB.W   D0,D2
               BEQ.B   Anie4
               DIVS.W  D2,D3
Anie4:         ADD.W   D1,D3
               MOVE.W  #XMAX,D2
Ary:           BSR.W   Drawline
Aend:          RTS
;-----

```

Rozdział XI

Optymalizacja szybkości działania programów

W tym rozdziale podaję sposoby przyspieszenia (optymalizacji) działania procedur. Często bowiem okazuje się, że zwykle, uniwersalne metody, nie wystarczają...

11.1 Zastępowanie instrukcji

a. *BRA/BSR xx*

Instrukcje *BRA/BSR xx* można zastąpić instrukcjami *BRA.S/BSR.S xx*, ale wtedy, gdy *xx* zawiera się w PC.

b. *MOVE.X #0*

Rozkazy *MOVE.X #0* można zastąpić instrukcjami *CLR.X*, *MOVEQ*, *SUBA.X*. Oto przykłady:

```
MOVE.L #0,count → CLR.L count
MOVE.L #0,d0 → MOVEQ #0,d0
MOVE.L #0,a0 → SUB.L a0,a0
```

c. *CLR.L Dx*

Instrukcję *CLR.L Dx*, gdzie *Dx* jest dowolnym rejestrem danych, można zastąpić instrukcją *MOVEQ #0,Dx*, co odniesie identyczny skutek, z tym, że będzie po prostu szybsze.

d. *CMP #0*

Instrukcję *CMP #0* można zastąpić przez *TST*.

e. *MOVE.L #nn,Dx*

Jeżeli w instrukcji *MOVE.L #nn,Dx* dana ma rozmiar bajtu ($-128 \leq nn \leq 127$), to z powodzeniem możemy ją zmienić na *MOVEQ #nn,Dx*.

f. ADD/SUB.X #nn

Jeśli dana do dodania/odejmowania ma rozmiar 3 bitów ($1 \leq nn \leq 8$), to rozkaz ADD/SUB.X #nn można zmienić na: ADDQ/SUBQ.X #nn

g. JMP/JSR xx

Instrukcje skoków JMP/JSR xx można zastąpić przez BRA/BSR xx, ale tylko wtedy, gdy xx zawiera się w PC (różnica PC i adresu skoku nie jest większa od słowa).

h. JSR/BSR xx

Możemy zastąpić skoki do podprogramów, skokami bezwarunkowymi (JMP/BRA xx). Pamiętajmy jednak o odpowiedniej modyfikacji konstrukcji programu.

i. LSL/ASL #1/2,xx

Instrukcje LSL/ASL można bez problemu zastąpić dodawaniem. Oto przykład:

```
LSL/ASL #1,D0 → ADD D0,D0
LSL/ASL #2,D0 → ADD D0,D0
                ADD D0,D0
```

j. MULU/MULS #yy,xx

Instrukcję MULS/MULU #yy,xx, gdzie yy jest potęgą dwójki, możemy zastąpić instrukcjami ASL/LSL. Instrukcję ASL powinniśmy stosować wtedy, gdy chcemy otrzymać prawidłowy znak wyniku. Można wtedy mnożyć liczbę ujemną, ale nie można mnożyć przez ujemną. Oto przykład:

```
MULU #2,d0 → ASL #1,d0 → ADD d0,d0

MULS/MULU #2,xx → ASL/LSL #1,xx
MULS/MULU #4,xx → ASL/LSL #2,xx
MULS/MULU #8,xx → ASL/LSL #3,xx
MULS/MULU #16,xx → ASL/LSL #4,xx
MULS/MULU #32,xx → ASL/LSL #5,xx
MULS/MULU #64,xx → ASL/LSL #6,xx
MULS/MULU #128,xx → ASL/LSL #7,xx
MULS/MULU #256,xx → ASL/LSL #8,xx
```

Uwaga: po wykonaniu instrukcji MULS/MULU znaczniki będą inne niż po LSL/ASL.

k. DIVU/DIVS #yy,xx

Instrukcję DIVU/DIVS #yy,xx, gdzie yy jest potęgą dwójki, możemy zastąpić instrukcjami ASR/LSR. Instrukcję ASR powinniśmy stosować wtedy, gdy chcemy otrzymać prawidłowy znak wyniku. Można wtedy dzielić liczbę ujemną, ale nie można dzielić przez ujemną. Oto przykład:

DIVU #2,d0 → ASR #1,d0

DIVS/DIVU #2,xx → ASR/LSR #1,xx
 DIVS/DIVU #4,xx → ASR/LSR #2,xx
 DIVS/DIVU #8,xx → ASR/LSR #3,xx
 DIVS/DIVU #16,xx → ASR/LSR #4,xx
 DIVS/DIVU #32,xx → ASR/LSR #5,xx
 DIVS/DIVU #64,xx → ASR/LSR #6,xx
 DIVS/DIVU #128,xx → ASR/LSR #7,xx
 DIVS/DIVU #256,xx → ASR/LSR #8,xx

Uwaga: po wykonaniu instrukcji DIVS/DIVU znaczniki będą inne niż po LSR/ASR.

l. MOVEA.L #nn

Instrukcję MOVEA.L #nn możemy zastąpić instrukcją MOVEA.W #nn, jeśli nn jest długości słowa ($0 \leq nn \leq \$7fff$). MOVEA jest instrukcją rozszerzającą znak (sign-extending).

m. ADDA.X #nn

Szybszą metodą na dodanie danej natychmiastowej do danego rejestru adresowego, jest użycie instrukcji LEA nn(). Możemy to zastosować tylko wtedy, gdy $-\$8000 \leq nn \leq \$7fff$. Oto prosty przykład:

ADDA.L #800,A0 → LEA 800(A0),A0

n. LEA nn()

Instrukcję LEA nn() można zastąpić instrukcją ADDQ.W #nn, ale tylko wtedy, gdy dana ma długość 3 bitów ($1 \leq nn \leq 8$). Oto przykład:

LEA 6(A0),A0 → ADDQ.W #6,A0

o. \$0000nnnn.l

Jeśli adres, na którym chcemy operować, ma długość słowa ($0 \leq nnnn \leq \$7fff$), to stosujemy adres 16-bitowy — \$nnnn.w. Przykład:

MOVE.L 4,A6 → MOVE.L 4.w,A6

Uwaga: adres nnnn jest rozszerzany znakowo do długiego słowa.

p. MOVE.L #xx,Ay

Rozkaz MOVE.L #xx,Ay możemy zastąpić instrukcją LEA xx,Ay lub LEA xx(PC),Ay.

q. MOVE.L Ax,Ay ; ADD #nnnn,Ay

Rozkaz: LEA nnnn(Ax),Ay może zastąpić sekwencję rozkazów: MOVE.L Ax,Ay; ADD #nnnn,Ay.

r. *MOVE.X nnnn ; LEA nnnn*

Aby przyspieszyć działanie procedur, możemy używać trybu adresowania względem PC. Oto przykłady:

```

MOVE.X nnnn -> MOVE.X nnnn(pc)
LEA nnnn    -> LEA nnnn(pc)      ; jeśli nnnn zawiera się w PC.
(Ax,Dx.l)   (Ax,Dx.w)           ; jeśli 0≤Dx≤$7fff

```

11.2 Sztuczki i triki

Instrukcje BSET, BCLR, itp., można zastąpić instrukcjami logicznymi. Oto odpowiednie wzory:

Instrukcja	Zastępowana przez	Warunek
BSET #xx,yy	ORI.W #2^xx,yy	0≤xx≤15
BCLR #xx,yy	ANDI.W #~(2^xx),yy	0≤xx≤15
BCHG #xx,yy	EORI.W #2^xx,yy	0≤xx≤15
BTST #xx,yy	ANDI.W #2^xx,yy	0≤xx≤15

Najlepiej, jeśli yy będzie rejestrem danych.

Uwaga: znaczniki po zastosowaniu alternatywnej instrukcji, będą różniły się od instrukcji manipulacji bitami.

11.3 Optymalizacja czasowa pętli

Przykład: przypuśćmy, że chcesz wykonać funkcję XOR z wartością w D0, na 4096 bajtach danych.

Normalnie wyglądałoby to tak:

```

Loop:  MOVE.W    #4096-1,d7
        EORI.B   d0,(a0)+
        DBF      d7,Loop

```

Wykonanie tej pętli, zabierze sporo czasu. Ale co powiedziałbyś na to:

```

Loop:  MOVE.W    #4096/4-1,d7
        EOR.L    d0,(a0)+      ; D0 zawiera ten sam bajt, powtórzony 4 razy
        DBF      d7,Loop

```

Również xor-ujemy 4096 bajtów! Jednak wykonujemy pętlę 1024 razy. Ale można jeszcze szybciej:

```

Loop:  MOVE      #4096/4/4-1,d7
        EOR.L    d0,(a0)+
        EOR.L    d0,(a0)+
        EOR.L    d0,(a0)+
        EOR.L    d0,(a0)+
        DBF      d7,Loop

```

Ta pętla jest najszybsza w porównaniu z poprzednimi. Można jeszcze ją przyspieszyć, dodając 8 lub 16 instrukcji **EOR.L** i odpowiednio zmniejszając licznik pętli. Można również całkowicie zrezygnować z pętli i postawić 1024 instrukcji **EOR.L**, jedną po drugiej. Jednak tę metodę należy stosować tylko w wypadku, gdy program musi błyskawicznie wykonywać daną operację.

11.4 Szybkie czyszczenie i wypełnianie pamięci

Odwiecznym problemem koderów, napotykanym przy pisaniu różnego rodzaju procedur „wektorówki” wypełnianej, jest szybkie czyszczenie pamięci ekranu. Oczekiwanie na zakończenie kasowania ekranu przez blittera, aby potem móc narysować kolejną fazę animacji, znacznie wydłuża działanie programu. Można jednak część (lub całość) ekranu wyczyścić mikroprocesorem. Jednak jak to zrobić szybko i skutecznie?

```
MOVE.L A7,TempSp ;przechowujemy zawartość rejestru A7 (wskaźnika stosu)
LEA MemEnd,A7 ;w A7 podajemy adres końca pamięci do wyczyszczenia
MOVEQ #0,D0 ;zerujemy wszystkie rejestry danych
;...dla wszystkich 7 rejestrów danych...
MOVEQ #0,D7
MOVE.L D0,A0 ;zerujemy wszystkie rejestry adresowe
;...dla wszystkich 7 rejestrów adresowych...
MOVE.L D0,A6
```

Po wykonaniu tych czynności inicjujących, możemy jedną instrukcją wyczyścić na raz 60 bajtów pamięci (15*4):

```
MOVEM.L D0-D7/A0-A6,-(A7)
```

Teraz powtarzamy tę instrukcję (byle nie w pętli!!!), tyle razy by wyczyścić pamięć. W tym celu możemy skorzystać z prostego wzoru:

$$\text{ilość powtórzeń} = \frac{\text{pamięć}}{60}$$

Po wyczyszczeniu pamięci, musimy pamiętać o przywróceniu poprzedniej wartości rejestru wskaźnika stosu. Piszemy więc:

```
MOVE.L TempSp,A7
```

Jeśli jednak mimo wszystko chciałbyś wykonywać instrukcję czyszczenia w pętli (za pomocą rejestru danych), to jedną instrukcją czyściłbyś tylko 56 bajtów.

11.5 Główne zasady pisania szybkich procedur.

- Jeśli optymalizujesz program, to najpierw znajdź jego części krytyczne czasowo: pętle (wykonywane kilkadziesiąt, czy kilkaset razy), procedury obsługi przerwań, itp. Ocenia się, że stanowią one najwięcej 10% kodu programu, ale odpowiadają za około 90% czasu wykonywania całej procedury.

- Nigdy nie optymalizuj części programów inicjujących i obsługi wyjścia z procedury (ang. *startup/exit code*). Nie staraj się także tego robić w wypadku procedur, które mają być wywołane tylko raz w całym programie.
- Jeśli zależy Ci na szybkości działania blittera, to ustaw bit BLTPRI w rejestrze DMACON (#10 w \$dff90a). Jeśli jednak bardziej zależy Ci na szybkości obliczeń, niż na operacjach blittera, to skasuj ten bit.
- Staraj się maksymalnie wykorzystać rejestry mikroprocesora! Jeśli będziesz wyniki pośrednie przechowywał i pobierał z pamięci, to znacznie wydłużysz czas wykonywania programu.
- Jeśli posiadasz wystarczająco dużo pamięci, to stwórz tablice mnożeń i dzielen, przekalkuluj wszystko co się da, lub zastępuj te instrukcje prostszymi (na przykład LSR, ADD)
- Jeśli piszesz procedurę stawiającą 12000 punktów szalejących na ekranie, to pisz procedurę ciurkiem, z jak najmniejszą ilością skoków, itp. W tej sytuacji staje się konieczne tablicowanie wyników instrukcji (na przykład LSR).
- Nie obciążaj zbytnio kanałów DMA! Przy włączonych 4 bitplanach (na Amidze 500) w Hiresie, obsługa staje się niesamowicie wolna. O szybkiej wektorówce w tym trybie można wtedy już tylko marzyć.

Rozdział XII

Układy AGA

Nie tak dawno firma Commodore wypuściła na rynek trzy nowe modele: Amigę 4000, 1200 i CD32. Komputery te zostały wyposażone w zupełnie nowe układy graficzne (oznaczone AGA), dające znacznie większe możliwości w porównaniu z układami montowanymi w poprzednich wersjach. Jednak firma postanowiła nie wypuszczać opisu rejestrów sprzętowych nowych układów, pozostawiając jedynie możliwość kontroli układów poprzez system operacyjny. Wiadomo jednak, że pisząc dema czy gry, odwoływanie się do systemu jest po prostu za wolne. Wobec tego, pojawił się piracki opis układów AGA.

12.1 Co nowego oferują układy AGA?

- Maksymalna ilość bitplanów została podwyższona do ośmiu i dostępna jest we wszystkich możliwych rozdzielczościach.
- Większa paleta — liczba rejestrów kolorów została rozszerzona do 256 i wykorzystuje 24-bitową paletę (przypada po 8 bitów na każdą składową koloru (RGB)). Korzystanie z 256 rejestrów kolorów możliwe jest w każdej rozdzielczości. Także paleta 16 777 216 kolorów może być wykorzystywana bez względu na tryb wyświetlania.
- Korzystanie z 256 24-bitowych rejestrów kolorów jest możliwe za pomocą tzw. banków.
Rozszerzony obsługa trybu Dual Playfield — możliwe jest teraz wyświetlanie dwóch playfieldów, z których każdy może mieć maksymalnie po 4-bitplany. Oczywiście dotyczy to każdej rozdzielczości.
- Większe możliwości scrollowania sprzętowego — BPLCON1 zawiera teraz wartość 8-bitowego przesunięcia dla każdego playfieldu. Scroll ten odbywać się może co 1 punkt superhiresu. Oznacza to, że w wypadku wyświetlania obrazka w lo-resie, możemy go przesunąć o 1/4 piksela w tym trybie.
- Rozszerzona obsługa spriteów — rozdzielczość spriteów może być ustawiona na lores, hires lub super-hires, niezależnie od rozdzielczości playfieldu. Połączone spritey są teraz dostępne we wszystkich trybach. Nieparzyste i parzyste spritey mogą używać niezależnych 16-to kolorowych banków z palety 256 kolorów. Mogą mieć one teraz szerokość 16, 32 lub 64 punktów. Opcjonalnie mogą być wyświetlane na ramce ekranu. Rozdzielczość pozioma pozycji spritea została zwiększona do 35ns (co odpowiada jednemu punktowi w super-hiresie)

- Tryb scan doubling — 15 kHz bitplany i spritey mogą teraz być wyświetlane bez drgań na monitorach 31 kHz (na przykład VGA).
- Istnieje możliwość kontroli rozmiaru pobieranych danych bitplaneów z BPLxDAT. W Amidze 500 DMA pobierało do wyświetlenia na raz tylko jedno słowo. Układy AGA oferują nam pobieranie 2, 4 lub 8 bajtów na raz.

12.2 Nowe tryby graficzne

Tryb	Liczba bitplanów	Liczba kolorów	Bandwidth ¹
LORES (320×200)	6	64	1
	7	128	1
	8	256	1
	8	HAM 262 144 ²	1
HIRES (640×200)	5	32	2
	6	EHB 64 ³	2
	6	HAM 4096 ⁴	2
	6	64	2
	7	128	2
	8	256	2
	8	HAM 262 144 ²	2
SUPERHIRES (1280×200)	1	2	1
	2	4	1
	3	8	2
	4	16	2
	5	32	4
	6	EHB 64 ³	4
	6	HAM 4096 ⁴	4
	6	64	4
	7	128	4
	8	256	4
	8	HAM 262 144 ²	4

Tryb	Liczba bitplanów	Liczba kolorów	Bandwidth ¹
VGA (160×480, 320×480, 640×480)	1	2	1
	2	4	1
	3	8	2
	4	16	2
	5	32	4
	6	EHB 64 ³	4
	6	HAM 4096 ⁴	4
	6	64	4
	7	128	4
	8	256	4
	8	HAM 262 144 ²	4

- ¹ Liczba Bandwidth określa minimalną ilość słów jaką DMA musi pobierać, by został wyświetlony odpowiedni tryb. Dla przykładu dla obrazu VGA w 32 kolorach DMA musi pobierać dwa długie słowa, podczas gdy dla 2 kolorów potrzeba już tylko jednego słowa. Ta zmiana umożliwia przesyłanie danych 4 razy szybciej...
- ² Dodany został nowy tryb graficzny: 8-mio bitplanowy HAM (HAM8). Sześć bitów przypada na zmianę składników koloru (lub numer koloru z palety głównej), a 2 bity służą jako bity kontrolne. Wszystkie tryby HAM-u (także stary HAM) można włączać we wszystkich rozdzielczościach (a nie tylko w LORES tak jak przedtem).
- ³ Tryb EHB jest również dostępny w nowych rozdzielczościach. Używa on 5 bitów do określenia rejestru koloru, określającego barwę danego punktu, a szóstego bitu dla 32 kolorów przyciemnionych o połowę.
- ⁴ Tryb HAM jest dostępny również w nowych rozdzielczościach.

12.3 Omówienie nowych możliwości

12.3.1 Bitplany

Układy AGA umożliwiają wykorzystanie 8 bitplanów. W pojedynczym plafieldzie można wykorzystać wtedy 256 kolorów. Wszystkie bitplany są dostępne w każdej rozdzielczości. Tak samo tryb DPM (dual playfield) jest dostępny we wszystkich rozdzielczościach, każdy w maksymalnie 16 kolorach. Bity od 8 do 15 w rejestrze BPLCON4 zawierają 8 bitową maskę dla 8 adresów bitplanów, XOR-ują pojedyncze bity. Pozwala to copperowi na zmianę palety kolorów

pojedynczą instrukcją. BPLCON1 zawiera teraz wartość 8-bitowego przesunięcia dla playfieldów. Rozdzielczość scrolla została powiększona do 35ns, co odpowiada 1 pikselowi w trybie SHRES. Bity BPAGEM, BPL32 w rejestrze FMODE odpowiadają za rozmiar danych pobieranych do wyświetlania. Może to być 2, 4 lub 8 bajtów. Jednak w związku z tymi zmianami, zmodyfikowane zostało także modulo (rejestr BPLMOD). W zależności od tego, ile pobieranych jest na raz bajtów, o tyle musimy zmniejszyć modulo. I tak:

FMOPDE	M
xxx3	-8
xxx2	-4
xxx1	-4
xxx0	0

Jeśli chcemy np. wyświetlić obrazek szerokości 320 punktów w 256 kolorach i trybie RAW-BLIT, a FMODE wynosi 3, to modulo będzie wynosić $7 \cdot 40 - 8$.

Dodany został nowy tryb graficzny: 8-mio bitplanowy HAM (tzw. HAM8). Sześć bitów przypada na zmianę składników koloru (lub numer koloru z palety głównej), a 2 bity służą jako bity kontrolne. Wszystkie tryby HAM-u (także stary HAM) można włączać we wszystkich rozdzielczościach (a nie tylko w LORES tak jak przedtem). Stary, 6-cio bitplanowy HAM działa także w trybie HIRES i SHRES (Super HIRES). W starym HAM-ie bitami kontrolnymi były bity z 5-go i 6-go bitplanu

bitplan 6	bitplan 5	Red	Green	Blue
0	0	(1 – 16) kolor z palety głównej		
0	1	zachowany	zachowany	wartość (BP1–BP4)
1	0	wartość (BP1–BP4)	zachowany	zachowany
1	1	zachowany	wartość (BP1–BP4)	zachowany

Nowy HAM8 działa troszkę inaczej. Tryb zostaje włączony, gdy włączymy 8 bitplanów w BPLCON0 i bit HOMEN. Bitplany 1 i 2 w tym trybie mają analogiczną funkcję jak bitplany 5 i 6 w trybie HAM6:

bitplan 1	bitplan 0	Red	Green	Blue
0	0	(1 – 64) kolor z palety głównej		
0	1	zachowany	zachowany	wartość (BP3–BP8)
1	0	wartość (BP3–BP8)	zachowany	zachowany
1	1	zachowany	wartość (BP3–BP8)	zachowany

Należy zauważyć, że modyfikacji ulega tylko 6 najwyższych bitów każdej składowej koloru. Pozostałe 2 bity (najmłodsze), pozostają bez zmian, to znaczy są takie jakie zostały ustawione na przykład przez kolor z palety głównej (64-kolorowej). Wynika z tego, że bez udziału kolorów z palety głównej mamy do dyspozycji 262144 kolorów ($64 \cdot 64 \cdot 64$ ($2^6 \cdot 2^6 \cdot 2^6$)).

Dla zachowania kompatybilności, pozostawiono tryb EHB (Extra Half Bright — 64 kolory). Włącza się go tak jak zwykle, czyli: SHRES, HIRES, HOMEN i DPF = 0, a liczba bitplanów wynosi 6. W rejestrze BPLCON2 mamy do dyspozycji bit (KILLEHB), który wyłącza ten tryb i mamy wtedy do dyspozycji pełną paletę 64 kolorów.

12.3.2 Spritey

Spritey mogą być teraz wyświetlane w trybach LORES, HIRES i SHRES niezależnie od rozdzielczości rysunku (tła). Także połączone spritey mogą być wyświetlane w każdym trybie.

Aby zmienić rozdzielczość sprite'ów, należy użyć bitów 5 i 7 w rejestrze \$dff106 (BPLCON3)

bit 7	bit 6	Rozdzielczość
0	0	Normalny ECS (LORES/HIRES = 140 ns, SUPERHIRES = 70 ns)
0	1	Zawsze LORES (140 ns)
1	0	Zawsze HIRES (70 ns)
1	1	Zawsze SUPERHIRES (35 ns)

Bity 3 i 2 w rejestrze FMODE odpowiadają za rozmiar pobieranych na raz danych do wyświetlenia sprite'ów, analogicznie jak dla bitplanów. Tym samym określają także maksymalną szerokość sprite'ów (16,32, lub 64 bity).

bit 3	bit 2	Szerokość	Słowa kontrolne
0	0	16 pikseli	2 słowa (normalne)
1	0	32 piksele	2 długie słowa
0	1	32 piksele	2 długie słowa
1	1	64 piksele	2 podwójne długie słowa

Należy jednak pamiętać, że:

- adres spritea o szerokości 16 pikseli musi być podzielny przez 2,
- adres spritea o szerokości 32 pikseli musi być podzielny przez 4,
- adres spritea o szerokości 64 pikseli musi być podzielny przez 16.

Tak wygląda struktura spritea o szerokości 16 punktów:

```
słowo C1, słowo C2
słowo A1, słowo B1
.
.
.
słowo An, słowo Bn
$0000 0000
```

- C1 — pierwsze słowo kontrolne,
- C2 — drugie słowo kontrolne,
- Ai, Bi — tworzą kształt i kolory spritea w danej linii.

Oto struktura spritea o szerokości 32 punktów:

```
        CNOP      0,8                ;informacja dla programu asemblerującego,
                                      ;że następny adres ma być podzielny przez 16
SPRITE32:
VSTART:  DC.B 0                ;długie słowo C1
HSTART:  DC.B 0
          DC.W 0
VSTOP:   DC.B 0,0              ;długie słowo C2
          DC.W 0
długie słowo A1, długie słowo B1
.
.
.
długie słowo An, długie słowo Bn
          DC.W 0,0,0,0          ;Koniec struktury spritea
```

- C1 — pierwsze długie słowo kontroli — pierwsze słowo kontroli, jest zawarte w wyższym słowie C1, niższe słowo C1 musi zawierać drugie słowo kontroli,
- C2 — drugie długie słowo kontroli — drugie słowo kontroli, jest zawarte w wyższym słowie C2, niższe słowo C2 musi być równe \$0000,
- Ai, Bi — tworzą kształt i kolory spritea w danej linii.

Struktura spritea o szerokości 64 punktów powinna wyglądać następująco:

```
        CNOP      0,8
SPRITE64:
VSTART:  DC.B 0                ;podwójne długie słowo C1
HSTART:  DC.B 0
          DC.W 0
          DC.L 0
VSTOP:   DC.B 0,0              ;podwójne długie słowo C2
          DC.W 0
          DC.L 0
podwójne długie słowo A1, podwójne długie słowo B1
.
.
.
podwójne długie słowo An, podwójne długie słowo Bn
          DC.W 0,0,0,0,0,0,0,0 ;Koniec struktury spritea
```

- C1 — pierwsze podwójne długie słowo kontroli (C1=W3:W2:W1:W0):
W3 jest pierwszym słowem kontroli,
W2 i W1 zawierają drugie słowo kontroli,

- C2 — drugie podwójne długie słowo kontroli (C2=W3:W2:W1:W0):
 W3 jest drugim słowem kontroli,
 Ai, Bi — tworzą kształt i kolory spritea w danej linii.

12.3.2.1 Poruszanie spritem z precyzją do 1/4 punktu.

Aby poruszać spritem o 1/4 piksela w LORES-ie (czyli 1 punkt w SUPERHIRES-ie), należy użyć bitów 3 i 4 w drugim słowie kontroli:

- bit 0 drugiego słowa kontroli=bit 2 pozycji poziomej,
- bit 3 drugiego słowa kontroli=bit 0 pozycji poziomej,
- bit 4 drugiego słowa kontroli=bit 1 pozycji poziomej.

12.3.2.2 Zmiana palety kolorów spriteów

Wszystkie spritey używają do wyświetlania jednej palety 16 kolorów. Oto jak przypisane są rejestry:

Spritey	Kolory
0 – 1	00 – 03
2 – 3	04 – 07
4 – 5	08 – 11
6 – 7	12 – 15

Teraz dla nieparzystych spriteów możesz używać innej palety, niż dla spriteów parzystych. 256 rejestrów kolorów dają 16 „sekcji” po 16 kolorów. Spriteom parzystym i nieparzystym możesz przydzielić różne „sekcje”. Oto jak będą wtedy przydzielane kolory:

Spritey	Kolory
0	00 – 03 palety spriteów parzystych
2	04 – 07 palety spriteów parzystych
4	08 – 11 palety spriteów parzystych
6	12 – 15 palety spriteów parzystych
1	00 – 03 palety spriteów nieparzystych
3	04 – 07 palety spriteów nieparzystych
5	08 – 11 palety spriteów nieparzystych
7	12 – 15 palety spriteów nieparzystych

Bity od 4 do 7 z rejestru \$dfff10c są używane do wyboru numeru „sekcji” palety kolorów, używanych przy wyświetlaniu spriteów parzystych. Natomiast bity od 0 do 3 służą do tych samych celów, co bity 4 – 7, ale dotyczą spriteów nieparzystych.

I tak dla spriteów parzystych:

bit 7	bit 6	bit 5	bit 4	Pierwszy kolor w paletce spriteów
0	0	0	0	\$0180/sekcja 0 (kolor 0)
0	0	0	1	\$01a0/sekcja 0 (kolor 15)
0	0	1	0	\$0180/sekcja 1 (kolor 31)
0	0	1	1	\$01a0/sekcja 1 (kolor 47)
0	1	0	0	\$0180/sekcja 2 (kolor 63)
0	1	0	1	\$01a0/sekcja 2 (kolor 79)
0	1	1	0	\$0180/sekcja 3 (kolor 95)
0	1	1	1	\$01a0/sekcja 3 (kolor 111)
1	0	0	0	\$0180/sekcja 4 (kolor 127)
1	0	0	1	\$01a0/sekcja 4 (kolor 143)
1	0	1	0	\$0180/sekcja 5 (kolor 159)
1	0	1	1	\$01a0/sekcja 5 (kolor 175)
1	1	0	0	\$0180/sekcja 6 (kolor 191)
1	1	0	1	\$01a0/sekcja 6 (kolor 207)
1	1	1	0	\$0180/sekcja 7 (kolor 223)
1	1	1	1	\$01a0/sekcja 7 (kolor 239)

Druą tabela jest dla bitów 0 do 3 (nieparzyste) i 4 do 7 (nieparzyste) rejestru \$dfff10c.

Bit SSCAN2 w FMODE włącza tryb scan-doubling dla spriteów. Kiedy jest włączony, to dla każdego spritea bit SH10 w SPRxPOS określa czy on też nie ma być w trybie scan-doubling. BPLCON3 zawiera poszczególne bity dotyczące zachowania się spriteów. Bity SPRES1 i SPRES0 kontrolują rozdzielczość spriteów (tzn. LORES, HIRRES, SHRES). Ustawienie bitu BRDRSPRT powoduje, że spritey są widoczne w obszarze ramki ekranu (na ramce). Bity od ESPRM4 do ESPRM7 pozwalają na przydzielenie palety kolorów dla parzystych, natomiast bity od OSPRM4 do OSPRM7 dla nieparzystych spriteów. W wyniku tego, połączone spritey używają bitów OSPRM.

Uwaga: pozycje pionowe spritea, START i STOP muszą być albo parzyste, albo nieparzyste (wysokość spritea musi być parzysta).

12.4 Tabela adresowania rejestrów kolorów

Ilość rejestrów kolorów została rozszerzona z 32-óch 12-sto bitowych rejestrów do 256-u 24-ro bitowych. W celu zapewnienia kompatybilności z wcześniejszymi wersjami Amig, zmodyfikowano nieco dostęp do palety kolorów.

Dodane zostały 3 nowe bity w rejestrze BPLCON3 (BANK0, BANK1, BANK2), które odpowiadają za bankowanie 32 rejestrów kolorów. To znaczy nie dodano nowych (adresowo) rejestrów kolorów, tylko te same rejestry przez zmianę banków kolorów odpowiadają za inne kolory, na przykład rejestr COLOR00 (\$180) odpowiada za kolor nr 00, 32, 64, 96, 128, 160, 192, 224.

W BPLCON3 mamy także do dyspozycji bit LOCT, decydujący, którą połowę składowych kolorów (RGB) będziemy ładować do rejestru koloru.

Jeśli na przykład chcemy ustawić kolor tła: R — \$1b, G — \$d5, B — \$07 to musimy najpierw załadować do \$180 wyższe 12 bitów (\$1d0), ustawić bit LOCT i zapisać do \$180 młodsze 12 bitów (\$b57). Należy postępować w takiej, a nie innej kolejności ponieważ normalny zapis wyższych 12-stu bitów powoduje automatyczne skopiowanie go także do młodszych 12-stu bitów dla zachowania kompatybilności z wcześniejszymi maszynami. Jeśli więc zapiszemy do rejestru \$dff180 wartość \$2f5 (przy wyzerowanym bicie LOCT), to zostanie ona rozszerzona automatycznie do 24 bitów (w naszym wypadku będzie to \$22ff55).

Bity BANK2, BANK1, BANK0 powodują taki układ 32-óch rejestrów kolorów:

BANK2	BANK1	BANK0	Numer koloru w adresach \$180 – \$1be
0	0	0	kolor \$00 — kolor \$1f (0 do 31)
0	0	1	kolor \$20 — kolor \$3f (32 do 63)
0	1	0	kolor \$40 — kolor \$5f (64 do 95)
0	1	1	kolor \$60 — kolor \$7f (96 do 127)
1	0	0	kolor \$80 — kolor \$9f (128 do 159)
1	0	1	kolor \$a0 — kolor \$bf (160 do 191)
1	1	0	kolor \$c0 — kolor \$df (192 do 223)
1	1	1	kolor \$e0 — kolor \$ff (224 do 255)

Oto fragment copperlisty ustawiający kolor tła na \$123456:

```
DC.L $01060000
DC.L $01800135
DC.L $01060200
DC.L $01800246
```

12.5 Kompatybilność

T_PIN czyści wszystkie bity we wszystkich nowych rejestrach kości AGA. Oto lista tych rejestrów:

- BPLCON3,
- BPLCON4,
- CLXCON2,
- DIWHIGH,
- FMODE.

Bit ECSENA (formalnie ENBPLCN3) jest używany do wyłączenia tych bitów w rejestrze BPLCON3, do których wcześniej nie było dostępu i zmienia do poprzedniego stanu bity BRDRBLNK, BRDNTRAN, ZDCLKEN, EXTBLEN i BRDRSPRT.

Rejestr CLXCON2 jest zerowany przez zapis do CLXCON, więc w starych grach powinna poprawnie działać obsługa kolizji.

DIWHIGH jest czyszczony przez zapis do rejestrów DIWSTRT lub DIWSTOP. Jest to pozostałość z układów ECS Denise

12.6 Kolizje

Nowy rejestr CLXCON2 zawiera 4 nowe bity. ENBP7 i ENBP6 włącza kolejno bity dla nowych bitplanów 7 i 8. Podobnie MVBP7 i MPBP8 dopasowane do nowych możliwości. CLXDAT jest nie zmieniony.

12.7 Monitory

Układ AGA daje możliwość wyświetlania obrazu w programowanej częstotliwości. Umożliwia to podłączenie monitora 31 kHz (VGA, MULTISYNC), przez co uzyskujemy niedrgający obraz przy rozdzielczościach takich jak dawniej w trybie INTERLACE.

12.8 Przełączanie z 15 kHz na 31 kHz

Dodano nowe możliwości pomocne przy wyświetlaniu razem 15 kHz i 31 kHz-owych obrazów z tą samą częstotliwością wyświetlania 31 kHz. LISA może ogólnie włączyć spritea w rozdzielczościach LORES, HIRES lub SUPERHIRES.

12.9 Jak wykryć układy AGA ?

Co należy zrobić, by wykryć układy AGA? Wystarczy sprawdzić zawartość rejestru LISAIID. Oto przykładowa procedura:

```

MOVE.W    $dff07c,d0
CMPI.B    #$f8,d0
BNE.S     NotAGA           ;Czy mamy do czynienia z układami AGA?
MOVEQ     #39,D0
JSR       -$228(a6)        ;Otwarcie biblioteki graficznej w wersji 39
TST.L     D0               ;Problemy z otwarciem biblioteki?
BEQ.S     NotAGA          ;Jeśli biblioteka została otwarta, to nie jest
                           ;to komputer z układami AGA
MOVE.L     D0,A6           ;d0=v39 GfxBase, skopiowany in A6
ST.B      AGA              ;ustawiamy znacznik układów AGA

BRA.S     AGACHIPS

CNOP      0,4              ;wyrównanie do długiego słowa
NotAGA:
JSR       -$198(a6)        ;Otwarcie biblioteki bez sprawdzania jej wersji
                           ;(funkcja OldOpenLibrary)
TST.L     D0
BEQ.W     EXIT             ;Problems opening gfxlib, so exit!
MOVE.L     D0,A6           ;d0=GfxBase, copied in A6
AGACHIPS:
LEA        GFXBASE(PC),A0 ;
MOVE.L     A6,(A0)         ;Zachowaj bazę biblioteki Gfx pod etykietą
                           ;GFXBASE
.....

```

Dodatek I

Lista rejestrów w porządku alfabetycznym

Używane symbole:

- P — nowy rejestr układu Pandora,
 p — dodane lub zmieniane w trybie HIRES,
 H — nowy rejestr w trybie HIRES,
 h — dodane lub zmieniane w trybie HIRES,
 A — rejestr układu Agnus/Alice,
 D — rejestr układu Denise/Lisa,
 P — rejestr układu Paula,
 W — tylko do zapisu,
 R — tylko do odczytu,
 ER — early read — transfer DMA do RAM-u, z dysku lub z blittera.

Uwaga 1:

Adres przy nazwie rejestru oznacza przesunięcie względem bazy rejestrów sprzętowych — \$dff000. Oznacza to, że jeśli podany jest adres: ADKCON \$09e, to właściwy adres rejestru ADKCON będzie wynosił \$dff09e, czyli \$dff000 + \$ 9e.

Uwaga 2:

Opisane zostały tylko najważniejsze rejestry układu AGA. Zostało to spowodowane brakiem oryginalnej dokumentacji, a także dość zawiłą istotą niektórych z tych rejestrów.

<i>nazwa</i>	<i>adres</i>	<i>r/w</i>	<i>układ</i>	<i>opis</i>
ADKCON	\$09e	W	P	Zapis kontroli dysku i dźwięku
ADKCONR	\$010	R	P	Odczyt kontroli dysku i dźwięku

Bit	Nazwa	Przeznaczenie
15	SET/CLR	Bit kontrolny. Określa, czy bity zapisane jako 1 będą ustawiane, czy czyszczone.
14 – 13	PRECOM	Czas prekompensacji dysku: %00 — 0, %01 — 140 ns, %10 — 280 ns, %11 — 560 ns.

Bit	Nazwa	Przeznaczenie
12	MFPREC	1 — MFM precomp, 0 — GCR precomp.
11	UARTBRK	1 — stop UART
10	WORDSYNC	Uaktywnia synchronizację dla dysku
9	MSBSYNC	1 — włączona synchronizacja odczytu dysku na najstarszym bicie wejścia. Zwykle używane w formacie GCR.
8	FAST	Szybkość operacji dyskowych: 1 — 2 μ s na bit (MFM), 0 — 4 μ s na bit (MFM lub GCR).
7	USE3PN	Użyj kanału 3 do modulacji niczego (audio).
6	USE2P3	Użyj kanału 2 do modulacji okresu kanału 3.
5	USE1P2	Użyj kanału 1 do modulacji okresu kanału 2.
4	USE0P1	Użyj kanału 0 do modulacji okresu kanału 1.
3	USE3VN	Użyj kanału 3 do modulacji niczego.
2	USE2V3	Użyj kanału 2 do modulacji głośności kanału 3.
1	USE1V2	Użyj kanału 1 do modulacji głośności kanału 2.
0	USE0V1	Użyj kanału 0 do modulacji głośności kanału 1.

AUDxDAT \$0aa W P Dane kanału dźwiękowego nr x.

Rejestr ten zawiera 2 bajty danych, które są sekwencyjnie wysyłane do przetwornika C/A, a dalej, np. do wzmacniacza. Kontroler DMA automatycznie ładuje dane z CHIP RAM-u spod wskazanego adresu. Procesor główny, po wyłączeniu kanałów DMA audio, może także zapisywać dane bezpośrednio do tego rejestru. Po zakończeniu przesyłania danych, generowane jest przerwanie AUDIO.

AUDxLCH \$0a0 W A Adres danych dźwięku dla kanału nr x.

AUDxLCL \$0a2 W A

Para tych rejestrów zawiera adres początkowy danych DMA dla kanału dźwiękowego nr x. Po zawartość tych rejestrów jest buforowana, więc po zakończeniu odtwarzania dźwięku, nie jest ona zmieniona. Adres danych musi się znajdować w CHIP RAM-ie. Najmłodszy bit adresu jest ignorowany.

AUDxLEN \$0a4 W P Długość danych dla kanału nr x.

Rejestr ten określa liczbę słów danych dla DMA, w celu odtworzenia ich przez kanał nr x.

AUDxPER \$0a6 W P Okres dla kanału nr x.

Rejestr ten zawiera okres (tempo) przesyłania danych dla kanału x. Minimalny okres wynosi 123 takty, czyli 28,86 kHz.

AUDxVOL \$0a8 W P Głośność dla kanału nr x.

Zawartość tego rejestru określa natężenie dźwięku. 0 jest głośnością minimalną, a 64 maksymalną. Należy także pamiętać, że natężenie dźwięku zależy od amplitudy fali.

BLTAFWM \$044 W A pierwsze słowo maski blittera dla źródła A

BLTALWM \$046 W A drugie słowo maski blittera dla źródła A

Blitter dokonuje operacji AND z wartością zawartą w tych rejestrach oraz pierwszym i ostatnim słowem linii danych z kanału źródłowego A. 0 w danym bicie, nie przepuszcza odpowiedniego bitu w słowie danych. Maski powinny być wypełnione jedynekami przy wypełnianiu lub trybie liniowym.

BLTCON0 \$040 W A Zerowy rejestr kontrolny blittera

BLTCON1 h \$042 W A Pierwszy rejestr kontrolny blittera.

Te dwa rejestry służą do kontroli operacji blittera.

Tryb przenoszenia („normalny”)			Tryb liniowy		
Bit#	BLTCON0	BLTCON1	Bit#	BLTCON0	BLTCON
15	ASH3	BSH3	15	ASH3	BSH3
14	ASH2	BSH2	14	ASH2	BSH2
13	ASH1	BSH1	13	ASH1	BSH1
12	ASA0	BSH0	12	ASH0	BSH0
11	USEA	0	11	1	0
10	USEB	0	10	0	0
09	USEC	0	09	1	0
08	USED	0	08	1	0
07	LF7	0	07	LF7	DPFF
06	LF6	0	06	LF6	SIGN
05	LF5	0	05	LF5	OVF
04	LF4	EFE	04	LF4	SUL
03	LF3	IFE	03	LF3	SUL
02	LF2	FCI	02	LF2	AUL

Tryb przenoszenia („normalny”)			Tryb liniowy		
Bit#	BLTCON0	BLTCON1	Bit#	BLTCON0	BLTCON
01	LF1	DESC	01	LF1	SING
00	LF0	LINE (=0)	00	LF0	LINE (=1)

ASH — wartość przesunięcia dla źródła A,
 BSH — wartość przesunięcia dla źródła B,
 USEA — bit kontrolny do włączenia kanału A,
 USEB — bit kontrolny do włączenia kanału B,
 USEC — bit kontrolny do włączenia kanału C,
 USED — bit kontrolny do włączenia kanału D (kanału docelowego (DEST)),
 LF — funkcja logiczna,
 EFE — włącz tryb wypełniania exclusive,
 IFE — włącz tryb wypełniania inclusive,
 DES — bit kontrolny trybu descending,
 LINE — bit kontrolny trybu liniowego,
 SIGN — znacznik liczby dodatniej lub ujemnej,
 SING — linie dla wypełniania (pojedynczy bit na linię poziomą),
 SUD — czasami góra lub dół,
 SUL — czasami góra lub lewo,
 AUL — zawsze góra lub lewo.

BLTSIZE \$058 W A Start blittera i rozmiar (szerokość i wysokość)

Rejestr ten służy do określenia wielkości prostokątnego okna, na którym blitter ma wykonać operację (w trybie liniowym szerokość musi wynosić 2, a wysokość musi być wysokością linii). Zapis tego rejestru powoduje automatycznie rozpoczęcie operacji (start blittera).

bit#	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
	h9	h8	h7	h6	h5	h4	h3	h2	h1	h0	v5	v4	v3	v2	v1	v0

h — wysokość (ilość linii poziomych) — maksymalna wysokość, to 1024 linie.
 v — szerokość — maksymalnie 64 słowa (1024 punkty)

BLTSIZH h \$05e W A Szerokość obszaru operacji i start blittera

bit#	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
	x	x	x	x	x	h10	h9	h8	h7	h6	h5	h4	h3	h2	h1	h0

BLTSIZV h \$05c W A Wysokość obszaru operacji

bit#	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
	x	v14	v13	v12	v11	v10	v9	v8	v7	v6	v5	v4	v3	v2	v1	v0

Rejestry BLTSIZH i BLTSIZV są rejestrami „nowej wersji” blittera służącymi do przenoszenia większych obszarów. Oryginalny rejestr wielkości przenoszonego obszaru pozostawiono dla zachowania kompatybilności. BLTSIZV powinien być zapisany pierwszy, przed zapisem BLTSIZH, który powoduje wystartowanie blittera. BLTSIZV nie potrzebuje powtórnego zapisu tej samej wartości co przed wystartowaniem blittera, gdyż pozostaje niezmieniony. Maksymalny rozmiar blitu 32752 pikseli na 32767 linii.

Uwaga: bity oznaczone "x" powinny być wyzerowane dla zachowania późniejszej kompatybilności.

BLTxDAT W \$074 W A Rejestr danych dla źródła x

Rejestr ten przechowuje pojedyncze słowo danych dla źródła x (x = A, B, C), dla operacji blittera. Normalnie jest ładowany przez DMA, lecz może to także wykonać procesor. W trybie liniowym BLTADAT jest używany jako rejestr indeksowy i powinien zawierać \$8000. BLTBDAT zawiera wzór linii (\$ffff dla linii ciągłej).

BLTxMOD \$064 W A Moduło dla źródła x

Rejestr ten zawiera moduło dla źródła blittera (x = A, B, C) lub dla przeznaczenia (x = D). Moduło jest automatycznie dodawane do adresu przy końcu każdej linii w celu ustalenia nowego adresu następnej linii.

BLTxPTH \$050 W A Adres danych dla blittera (starsze słowo)

BLTxPTL \$052 W A Adres danych dla blittera (młodsze słowo)

Ta para rejestrów zawiera adres danych DMA dla źródła (x = A, B, C) lub przeznaczenia (x = D). Po zakończeniu operacji blittera, rejestr ten zawiera ostatni adres danej plus moduło.

BPLIMOD \$108 W A Moduło bitplanów nieparzystych

BPL2MOD \$10a W A Moduło bitplanów parzystych

Rejestry te zawierają moduło (ze znakiem) dla bitplanów nieparzystych i parzystych.

BPLCON0 p \$100 W A rejestr kontrolny bitplanów nr 0

BPLCON1 p \$102 W A rejestr kontrolny bitplanów nr 1

BPLCON2 p \$104 W A rejestr kontrolny bitplanów nr 2

BPLCON3 p \$106 W A rejestr kontrolny bitplanów nr 3

BPLCON4 p \$10c W A rejestr kontrolny bitplanów nr 4

Rejestry te służą do kontrolowania trybów wyświetlania playfieldu.

bit#	BPLCON0	BPLCON1	BPLCON2	BPLCON3	BPLCON4
15	HIRES	PF2H7 = 0	x	BANK2 = 0	BPLAM7 = 0
14	BPU2	PF2H6 = 0	ZDBPSEL2	BANK1 = 0	BPLAM6 = 0
13	BPU1	PF2H1 = 0	ZDBPSEL1	BANK0 = 0	BPLAM5 = 0
12	BPU0	PF2H0 = 0	ZDBPSEL0	PF2OF2 = 0	BPLAM4 = 0
11	HAM	PF1H7 = 0	ZDBPEN	PF2OF1 = 1	BPLAM3 = 0
10	DPF	PF1H6 = 0	ZDCTEN	PF2OF0 = 1	BPLAM2 = 0
9	COLOR	PF1H1 = 0	KILLEHB	LOCT = 0	BPLAM1 = 0
8	GAUD	PF1H0 = 0	RDRAM = 0	x	BPLAM0 = 0
7	UHRES	PF2H5	SOGEN = 0	SPRES1 = 0	ESPRM7 = 0
6	SHRES	PF2H4	PF2PRI	SPRES0 = 0	ESPRM6 = 0
5	BYPASS = 0	PF2H3	PF2P2	BRDRBLNK = 0	ESPRM5 = 0
4	BPU3 = 0	PF2H2	PF2P1	BRDNTRAN = 0	ESPRM4 = 1
3	LPEN	PF1H5	PF2P0	x	OSPRM7 = 0
2	LACE	PF1H4	PF1P2	ZDCLKEN = 0	OSPRM6 = 0
1	ERSY	PF1H3	PF1P1	BRDSPRT = 0	OSPRM5 = 0
0	ECSENA = 0	PF1H2	PF1P0	EXTBLKEN = 0	OSPRM4 = 1

- x — nie używane — wpisz 0 dla zachowania późniejszej kompatybilności,
 HIRES — tryb podwójnej rozdzielczości poziomej (640×256 lub 640×512 interlace),
 BPUx — liczba bitplanów 000 – 1000 (0 włącza 8 BPL),
 HAM — tryb Hold And Modify (HAM), używa 6 lub 8 bitplanów — nowy tryb HAM8 jest włączony gdy BPL = 1000,
 DPF — tryb DPM (Double Playfield) — pierwszy ekran zajmuje nieparzyste bitplany, drugi parzyste.

Jeśli BPU = 6, HAM = 0, DPF = 0, to jest włączony tryb EXTRA-HALFBRITE (EHB) — 32 kolory normalne + 32 o połowę ciemniejsze.

- COLOR — włącza kolor na wyjściu (A1000),
 GAUD — włączenie Genlock Audio,
 SHRES — tryb super hi-res (1280×256, 1280×512),
 BYPASS — bitplany są scrollowane i priorytetowane normalnie, lecz omijana jest tablica kolorów i 8-bitowa szerokość pokazywania danych na R(7-0),

- LPEN — włączenie pióra świetlnego,
 LACE — tryb Interlace — dwa razy więcej punktów w pionie,
 ERSY — zewnętrzna synchronizacja (HSYNC, VSYNC),
 ECSENA — jeśli 0 to bity BRDRBLNK, BRDNTRAN, ZDCLKEN, BRDSPRT i EXTBLKEN z BPLCON3 są wyłączone,
 PF2Hx — scroll poziomy nieparzystych BPL, $x = 0 - 7$,
 PF1Hx — scroll poziomy parzystych BPL, $x = 0 - 7$,
 KILLEHB — Wyłącza tryb extra half brite (EHB) dla uzyskania pełnych 64 kolorów,
 SOGEN — kiedy jest zapalony powoduje ustawienie na styku wyjścia SOG,
 PF2PRI — daje priorytet playfieldu 2 nad playfieldem 1,
 PF2Px — bity priorytetu playfieldu 2 (nad spriteami),
 PF1Px — bity priorytetu playfieldu 1 (nad spriteami),
 BANKx — wybór jednego z ośmiu banków koloru, $x = 0 - 2$,
 PF20Fx — ustala tablice kolorów kiedy playfield 2 ma priorytet w trybie dual playfield:

PF20F			Ustawione bitplany								OFFSET · (przesunięcie)
2	1	0	8	7	6	5	4	3	2	1	(dec)
0	0	0	—	—	—	—	—	—	—	—	0
0	0	1	—	—	—	—	—	—	1	—	2
0	1	0	—	—	—	—	—	1	—	—	4
0	1	1	—	—	—	—	—	1	—	—	8 (normalnie)
1	0	0	—	—	—	1	—	—	—	—	16
1	0	1	—	—	1	—	—	—	—	—	32
1	1	0	—	1	—	—	—	—	—	—	64
1	1	1	1	—	—	—	—	—	—	—	128

- LOCT — dyktuje które półbajty składowych kolorów będą ładowane do rejestru koloru — zapis normalny do wyższych 12-bitów powoduje automatyczny zapis tej samej wartości do młodszych 12-bitów dla zachowania kompatybilności,
 SPRESx — rozdzielczość wszystkich 8 spritów ($x = 0, 1$);

SPRES1	SPRES0	rozdzielczość spritea
0	0	ECS (LORES, HIRES = 140ns, SHRES = 70ns)
0	1	LORES (140ns)
1	0	HIRES (70ns)
1	1	SHRES (35ns)

- BRDRBLNK — „obszar ramki” jest czyszczony zamiast koloru (0) — wyłączone kiedy ECSENA = 0.
- BRDRSPRT — włącza spritey poza obszarem wyświetlanego okna — wyłączone kiedy ECSENA = 0.
- BPLAMx — to 8 bitowe pole jest XOR-owane z 8 bitplanowym adresem koloru,
- ESPRMx — to 4 bitowe pole określa z której „sekcji” palety kolorów mają korzystać spritey parzyste,
- OSPRMx — to 4 bitowe pole określa z której „sekcji” palety kolorów mają korzystać spritey nieparzyste.

BPLxDAT \$110 W D Dane bitplanu nr x

Rejestry te otrzymują dane DMA pobierane z RAM-u na podstawie rejestrów adresowych bitplanów. Mogą być zapisywane przez procesor.

BPLxPTH \$0e0 W A Adresy bitplanów (starsze słowo)

BPLxPTL \$0e2 W A Adresy bitplanów (młodsze słowo)

Para rejestrów BPLxPTH i BPLxPTL zawiera adres początkowy danych DMA dla bitplanu nr x. Rejestry te muszą być zapisywane przez copper lub procesor główny, co każde wygaszanie pionowe ekranu.

CLXCON \$098 W A Kontrola kolizji.

Rejestr ten określa, które bitplany są włączone do wykrywania kolizji oraz ich pożądany status. Wyłączone bitplany nie wykrywają kolizji. Jeśli jednak wszystkie bitplany będą włączone, to kontrola zawsze wykaże kolizję.

bit	Nazwa	Przeznaczenie
15	ENSP7	włącz sprite 7 (lub 6)
14	ENSP5	włącz sprite 5 (lub 4)
13	ENSP3	włącz sprite 3 (lub 2)
12	ENSP1	włącz sprite 1 (lub 0)
11	ENBP6	włącz bitplan 6
10	ENBP5	włącz bitplan 5
9	ENBP4	włącz bitplan 4
8	ENBP3	włącz bitplan 3
7	ENBP2	włącz bitplan 2
6	ENBP1	włącz bitplan 1
5	MVB6	wartość zgody dla kolizji z bitplanem 6
4	MVB5	wartość zgody dla kolizji z bitplanem 5

bit	Nazwa	Przeznaczenie
3	MVB4	wartość zgody dla kolizji z bitplanem 4
2	MVB3	wartość zgody dla kolizji z bitplanem 3
1	MVB2	wartość zgody dla kolizji z bitplanem 2
0	MVB1	wartość zgody dla kolizji z bitplanem 1

Zapis do tego rejestru powoduje wyczyszczenie bitów w CLXCON2 (układy AGA).

CLXCON2 P \$10c W D Rozszerzona kontrola kolizji.

Rejestr kontroli kolizji kiedy mamy włączone 7 lub 8 bitplanów (porównaj CLXCON).

bit	Nazwa	Przeznaczenie
15 – 08	—	nie używane
7	ENBP 8	włącz bitplan 8
6	ENBP 7	włącz bitplan 7
5 – 2	—	nie używane
1	MVBP 8	wartość zgody dla kolizji z bitplanem 8
0	MVBP 7	wartość zgody dla kolizji z bitplanem 7

CLXDAT \$00e R D Rejestr danych kolizji

Po odczytaniu tego rejestru, otrzymujemy aktualny status kolizji, a rejestr jest automatycznie czyszczony.

bit#	Zauważone kolizje
15	nie używany
14	sprite 4 (lub 5) i sprite 6 (lub 7)
13	sprite 2 (lub 3) i sprite 6 (lub 7)
12	sprite 2 (lub 3) i sprite 4 (lub 5)
11	sprite 0 (lub 1) i sprite 6 (lub 7)
10	sprite 0 (lub 1) i sprite 4 (lub 5)
9	sprite 0 (lub 1) i sprite 2 (lub 3)
8	playfield 2 i sprite 6 (lub 7)

bit#	Zauważone kolizje
7	playfield 2 i sprite 4 (lub 5)
6	playfield 2 i sprite 2 (lub 3)
5	playfield 2 i sprite 0 (lub 1)
4	playfield 1 i sprite 6 (lub 7)
3	playfield 1 i sprite 4 (lub 5)
2	playfield 1 i sprite 2 (lub 3)
1	playfield 1 i sprite 0 (lub 1)
0	playfield 1 i playfield 2

COLORxx \$180-1be W

Tablica xx kolorów

Są to 32 rejestry kolorów (xx = 00 – 31). Razem z bitami do bankowania tych rejestrów mają 256 lokacji do zapisu kolorów z palety (tylko AGA). Są to praktycznie lokacje z dwoma wartościami kontrolowanymi przez bit LOCT. Kiedy LOCT=0 zapisujemy 4 starsze bity składowych koloru RGB, oraz bit T (przezroczystość w Genlocku), zapis do tego rejestru powoduje automatyczny zapis do 4-ech młodszych bitów, które zmieniamy gdy LOCT = 1. Wyjaśnia to dokładniej poniższa tabelka.

Bit#	15 – 12				11 – 8				7 – 4				3 – 0			
LOCT=0	X	X	X	X	R7	R6	R5	R4	G7	G6	G5	G4	B7	B6	B5	B4
LOCT=1	X	X	X	X	R3	R2	R1	R0	G3	G2	G1	G0	B3	B2	B1	B0

T — TRANSPARENCY,

R — RED,

G — GREEN,

B — BLUE,

X — nie używane

Normalnie jednak można używać kolorów 12 bitowych dokonujemy tego, przez zapis danej wartości do rejestru, przy wyzerowanym bicie LOCT:

Bit#	15 – 12				11 – 8				7 – 4				3 – 0			
LOCT=0	X	X	X	X	R3	R2	R1	R0	G3	G2	G1	G0	B3	B2	B1	B0

COPILCH	h	\$080	W	A	Adres pierwszej copperlisty (starsze słowo)
COPILCL		\$082	W	A	Adres pierwszej copperlisty (młodsze słowo)
COP2LCH	h	\$084	W	A	Adres drugiej copperlisty (starsze słowo)
COP2LCL		\$086	W	A	Adres drugiej copperlisty (młodsze słowo)
COPCON	h	\$02c	W		Rejestr kontrolny coppera

Bit#	Nazwa	Funkcja
1	CDANG	Tryb bezpieczeństwa — umożliwia copperowi dostęp do wszystkich rejestrów blittera (CDANG = 1)

COPJMP1 \$088 S A Restart coppera przy pierwszej copperliście.

COPJMP2 \$08a S A Restart coppera przy drugiej copperliście.

Zapisanie dowolnego z tych rejestrów, spowoduje wykonywanie copperlisty coppera o początku zapisanym w rejestrach lokacji coppera. Copper może także zapisywać te rejestry poprzez copperlistę, powodując tym własny restart.

COPINS \$08c W A Identyfikacja pobranej instrukcji coppera

Jest to bufor służący do identyfikacji pobranej instrukcji coppera.

DDFSTOP \$094 W A Pobieranie danych: pozycja początkowa (pozioma).

DDFSTRT \$092 W A Pobieranie danych: pozycja końcowa (pozioma).

Ta para rejestrów kontroluje poziomy początek i koniec pobierania danych wyświetlania w czasie.

bit#	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
	x	x	x	x	x	x	x	x	H8	H7	H6	H5	H4	H3	x	x

Bity oznaczone znakiem „x” są nie używane i powinny być wyzerowane, dla zachowania późniejszej kompatybilności.

DDFSTRT (lewy kraniec poboru danych wyświetlania)

Zamiar	H8	H7	H6	H5	H4
maksymalny	0	0	1	0	1
szeroki	0	0	1	1	0
normalny	0	0	1	1	1
wąski	0	1	0	0	0

DDFSTOP (prawy kraniec poboru danych wyświetlania)

Zamiar	H8	H7	H6	H5	H4
wąski	1	1	0	0	1
normalny	1	1	0	1	0
szeroki	1	1	0	1	1

DIWSTOP \$090 W AD Koniec okna wyświetlania (pozycja pion.-pozioma lewego, górnego rogu).

DIWSTRT \$08e W AD Początek okna wyświetlania (pozycja pion.-pozioma prawego, dolnego rogu).

bit#	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
	V7	V6	V5	V4	V3	V2	V1	V0	H7	H6	H5	H4	H3	H2	H1	H0

DIWSTRT jest pionowo ograniczony do górnych 2/3 wyświetlania ($V8 = 0$), a poziomo do lewych 3/4 wyświetlania ($H8 = 0$).

DIWSTOP jest pionowo ograniczony do dolnej 1/2 wyświetlania, a poziomo do prawej 1/4 wyświetlania ($H8 = 1$).

Bity oznaczone znakiem „x” są nie używane i powinny być wyzerowane, dla zachowania późniejszej kompatybilności.

DIWHIGH \$1e4 W AD Dalsze bity pozycji okna wyświetlania.

bit#	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
	x	x	H10	H1	H0	V10	V9	V8	x	x	H10	H1	H0	V10	V9	V8
	(stop)								(start)							

W układach AGA w celu powiększenia możliwości definiowania rozmiaru okna wyświetlania, dodano dodatkowy rejestr odpowiedzialny z dodatkowe bity pozycji okna wyświetlania.

DMACON \$096 W ADP kontrola DMA-zapis

DMACONR \$002 R AP kontrola DMA (i status blittera)-odczyt

bit	Nazwa	Przeznaczenie
15	SET/CLR	Bit kontrolny. Określa, czy bity zapisane jako 1 będą ustawiane, czy czyszczone.
14	BBUSY	Status blittera. Jeśli wynosi 1, to blitter nie jest wolny.

bit	Nazwa	Przeznaczenie
13	BZERO	Bit stanu 0 logicznego dla blittera. Pozostaje 1, gdy w czasie operacji blittera, na wyjściu jest 0.
12 – 13		Nie używane.
10	BLTPRI	Priorytet blittera. Gdy wynosi 1, to blitter ma priorytet nad procesorem głównym.
9	DMAEN	Włączenie dostępu do całego DMA (1 — włączony).
8	BPLEN	DMA bitplanów (1 — DMA bitplanów włączone)
7	COPEN	DMA coppa (1 — DMA coppa włączone)
6	BLTEN	DMA blittera (1 — DMA blittera włączone)
5	SPREN	DMA spriteów (1 — DMA spriteów włączone)
4	DSKEN	DMA dysku (1 — DMA dysku włączone)
3	AUD3EN	DMA kanału dźwiękowego nr 3 (1 — włączone)
2	AUD2EN	DMA kanału dźwiękowego nr 2 (1 — włączone)
1	AUD1EN	DMA kanału dźwiękowego nr 1 (1 — włączone)
0	AUD0EN	DMA kanału dźwiękowego nr 0 (1 — włączone)

DSKBYTR \$01a R p

Rejestr ten jest buforem danych dysk-mikroprocesor. Przy odczycie dane z dysku ładowane są do tego rejestru po bajcie za jednym razem, a bit 15 jest ustawiony na 1 (DSKBYT).

bit	Nazwa	Przeznaczenie
15	DSKRYBT	Odczyt danych: kiedy 1, to oznacza, że rejestr zawiera bezbłędną daną z dysku
14	DMAON	To samo, co DMAEN w DSKLEN po wykonaniu operacji AND z bitem DMAEN w DMAON.
13	DISKWRITE	To samo, co WRITE w DSKLEN.
13	WORDEQUAL	Jeden, gdy rejestr DSKSYNC jest równy danej dyskowej.
11 – 8		Nie używane.
7 – 0	DATA	Bajt danej dyskowej DMA.

DSKDAT	\$026	W	P	Zapis danej dyskowej DMA.
DSKDATR	\$008	ER	P	Odczyt danej dyskowej DMA

Rejestr ten zawiera dwa bajty danych DMA, które są wysyłane (zapis) lub pobierane (odczyt) z dysku.

DSKLEN	\$024	W	P	Długość danych dyskowych
---------------	-------	---	---	--------------------------

bit	Nazwa	Przeznaczenie
15	DMAEN	Włączenie dostępu do DMA dysku
14	WRITE	Tryb zapisu dla dysku. Dane będą przesyłane z pamięci na dysk. Włączony, jeśli WRITE = 1.
13 – 0	LENGHT	Ilość słów do przesłania.

DSKPTH	h	\$020	W	A	pointer danych dyskowych w pamięci (starsze słowo)
DSKPTL		\$022	W	A	pointer danych dyskowych w pamięci (młodsze słowo)

Ta para rejestrów zawiera adres danych dyskowych DMA. W przypadku odczytu, dane będą z dysku przesyłane do pamięci pod wskazywany adres. W przypadku zapisu, będą one przenoszone spod tego adresu do kontrolera dysku. Adresy te muszą być inicjowane przez procesor główny, ewentualnie coppera, przed włączeniem dostępu do DMA dysku.

DSKSYNC	\$07e	W	P	rejestr synchronizacji dyskowej
----------------	-------	---	---	---------------------------------

Rejestr ten zawiera słowo synchronizacji, od którego może rozpocząć się przenoszenie danych z dysku. W tym celu należy zapisać do tego rejestru wartość znacznika synchronizacji i ustawić bit 10 w ADKCON.

FMODE	p	\$1fc	W	Kontrola pobierania danych
--------------	---	-------	---	----------------------------

bit	Nazwa	Przeznaczenie
15	SSCAN2	Włączenie trybu SCAN-DOUBLING dla sprite'ów.
14	BSCAN2	Włączenie trybu SCAN-DOUBLING dla bitplan'ów.
13 – 4		Nie używane.
3	SPAGEM	Podwójne pobieranie danych dla sprite'ów.
2	SPR32	Pojedyncza dana dla spritea będzie wynosić 32 bity.
1	SPAGEM	Podwójne pobieranie danych dla bitplan'ów.
00	SPR32	Pojedyncza dana dla bitplanu będzie wynosić 32 bity.

BPAGEM	BPL32	Pobieranie danych	Szerokość w punktach
0	0	Po 2 bajty	16
0	1	Po 4 bajty	32
1	0	Po 4 bajty	32
1	1	Po 8 bajtów	64

SPAGEM	SPR32	Pobieranie danych	Szerokość w punktach
0	0	Po 2 bajty	16
0	1	Po 4 bajty	32
1	0	Po 4 bajty	32
1	1	Po 8 bajtów	64

INTENA \$09a W P Bity włączenia przerwania (zapis)
INTENAR \$01c R P Bity włączenia przerwania (odczyt)

bit	Nazwa	Poziom	Przeznaczenie
15	SET/CLR		Bit kontrolny. Określa, czy bity zapisane jako 1 będą ustawiane, czy czyszczone.
14	INTEN		Włączenie dostępu do wszystkich przerwań.
13	EXTER	6	Przerwanie zewnętrzne.
12	DSKSYN	5	DSKSYNC zgadza się z daną dyskową.
11	RBF	5	Bufor portu szeregowego jest pełny.
10	AUD3	4	Blok kanału dźwiękowego 3 zakończony.
9	AUD2	4	Blok kanału dźwiękowego 2 zakończony.
8	AUD1	4	Blok kanału dźwiękowego 1 zakończony.
7	AUD0	4	Blok kanału dźwiękowego 0 zakończony.
6	BLIT	3	Blitter zakończył pracę.
5	VERTB	3	Początek wygaszania pionowego.

bit	Nazwa	Poziom	Przeznaczenie
4	COPER	3	Przerwanie zarezerwowane dla coppera. Może być wywoływane poprzez instrukcję coppera w copperliscie.
3	PORTS	2	Porty We/Wy i zegary.
2	SOFT	1	Zarezerwowane dla przerwań programowych.
1	DSKBLK	1	Blok dysku zakończony.
0	TBE	1	Pusty bufor transmisji portu szeregowego.

INTREQ \$09c W P Bity zgłoszenia przerwania (zapis)

INTREQR \$01e R P Bity zgłoszenia przerwania (odczyt)

Rejestry te służą do zgłoszenia przerwania lub do określenia źródła przerwania.

JOY0DAT \$00a R D Dane myszy/joysticka z portu 0

JOY1DAT \$00c R D Dane myszy/joysticka z portu 1

bit#	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
0DAT	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0	X7	X6	X5	X4	X3	X2	X1	X0
1DAT	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0	X7	X6	X5	X4	X3	X2	X1	X0

W celu wykrycia kierunku aktualnie wskazywanego przez joystick, należy skorzystać z poniższej tabeli:

Aby wykryć	Odczytaj następujące bity
Góra	Y1 xor Y0
Dół	X1 xor X0
Lewo	Y1
Prawo	X1

JOYTEST \$036 W D Natychmiastowy zapis do wszystkich 4 liczników myszy lub joysticka na raz.

Dane dla zapisu liczników:

bit#	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0DAT	Y7	Y6	Y5	Y4	Y3	Y2	X	X	X7	X6	X5	X4	X3	X2	X	X
1DAT	Y7	Y6	Y5	Y4	Y3	Y2	X	X	X7	X6	X5	X4	X3	X2	X	X

LISAID H \$07c R D Identyfikacja układu Denise/Lisa

Oryginalny Denise (8362) nie posiada tego rejestru. ECS Denise (8373) zwraca wartość \$fc w młodszych ośmiu bitach. Układ Lisa zwraca \$f8. Wyższe 8 bitów tego rejestru jest ładowane z szyny serial mouse bus i jest zarezerwowane dla przyszłych rozszerzeń.

POT0DAT \$012 R P Obsługa nietypowych urządzeń sterowniczych w porcie 0

POT1DAT \$014 R P Obsługa nietypowych urządzeń sterowniczych w porcie 1

bit	Nazwa	Przeznaczenie
15 – 8		Licznik pionowy joysticka analogowego lub prawego „wioselka”.
7 – 0		Licznik poziomy joysticka analogowego lub lewego „wioselka”.

POTGO \$034 W P Zapis danych portu dla tzw. pot.

POTGOR \$016 W P Odczyt danych portu dla tzw. pot.

Bit	Nazwa	Przeznaczenie
15	OUTRY	Udostępnienie wyjścia dla pinu 9 w game port 1.
14	DATRY	Aktualny stan pinu 9 w game port 1 (1 — napięcie dodatnie).
13	OUTRX	Udostępnienie wyjścia dla pinu 5 w game port 1.
12	DATRX	Aktualny stan pinu 5 w game port 1 (1 — napięcie dodatnie).
11	OUTLY	Udostępnienie wyjścia dla pinu 9 w game port 0.

Bit	Nazwa	Przeznaczenie
10	DATLY	Aktualny stan pinu 9 w game port 0 (1 — napięcie dodatnie).
9	OUTLX	Udostępnienie wyjścia dla pinu 5 w game port 0.
8	DATLX	Aktualny stan pinu 5 w game port 0 (1 — napięcie dodatnie).
7 – 1		Nie używane.
0	START	Bit uruchamiający dla liczników pot.

REFPTR \$028 W A Odświeżanie pointera.

Rejestr ten jest używany jako generator adresowy odświeżania dynamicznego RAM-u. Jego zapis służy przeważnie celom testowym.

SERDAT \$030 W P Wyjście szeregowe danych.

bit	Nazwa	Przeznaczenie
15 – 10		Nie używane.
9	STP	Bit zatrzymujący (przy wysłaniu 9 bitów danych).
8	DB8 lub STP	Dziewiąty bit danych lub bit zatrzymujący.
7 – 0	DB7 – DB0	Bajt danych przeznaczony do transmisji.

SERDATR \$018 R P Status portu szeregowego.

Bit	Nazwa	Przeznaczenie
15	OVRUN	Bufor odbioru jest przepełniony.
14	RBF	Bufor odbioru jest pełny.
13	TBE	Bufor nadawania jest pusty.
12	TSRE	Rejestr przesunięcia nadawania jest pusty.
11	RXD	Aktualny stan linii RXD portu szeregowego (pin 3).
10		Nie używany.
9	STP	Bit zatrzymujący (przy pobieraniu 9 bitów danych).

Bit	Nazwa	Przeznaczenie
8	DB8 lub STP	Dziewiąty bit danych lub bit zatrzymujący.
7 – 0	DB7 – DB0	Bajt danych przeznaczony do transmisji.

SERPER \$032 W P Okres portu szeregowego i kontrola.

Rejestr ten określa szybkość przesyłania danych poprzez port szeregowy. Bity 0 – 14 odpowiadają za okres, a bit 15 za liczbę bitów danych oczekiwanych przy odbiorze (1 — dziewięć, 0 — osiem bitów). W celu obliczenia okresu, użyj następującego wzoru:

$$\text{okres} = \frac{3579546}{\text{szybkość transmisji}} - 1$$

Szybkość transmisji określona jest w bitach na sekundę. Poniższa tabelka prezentuje okresy dla sześciu najbardziej popularnych prędkości przesyłania danych:

Szybkość transmisji	Wartość okresu
300	11931
600	5965
1200	2982
2400	1490
4800	745
9600	372

SPRxCTL p \$142 W AD Pozycja pion. końca spritea i kontrola.

SPRxPOS \$140 W AD Pozycja pion.–pozioma początku spritea.

Te dwa rejestry współpracują razem jako rejestry kontrolne pozycji, rozmiaru i cech spritea. Zazwyczaj są ładowane przez spriteowe kanały DMA podczas wygaszania pionowego, lecz mogą być także ładowane przez procesor główny lub copper.

SPRxPOS:

bit #	Nazwa	Przeznaczenie
15 – 8	SV7 – SV0	Wartość pionowa początku.
7 – 0	SH8 – SH1	Wartość pozioma początku.

SPRxCTL:

bit #	Symbol	Przeznaczenie
15 – 8	EV7 – EV0	Młodsze 8 bitów pozycji pionowej końca.
7	ATT	Bit kontrolny łączenia spriteów.
6!	SV9	Dziesiąty bit pozycji pionowej początku.
5!	EV9	Dziesiąty bit pozycji pionowej końca.
4!	SH1 = 0	Drugi bit pozycji poziomej (1/2 punktu w lo-resie).
3!	SH0 = 0	Pierwszy bit pozycji poziomej (1 punkt w super-hiresie).
2	SV8	Dziewiąty bit pozycji pionowej początku.
1	EV8	Dziewiąty bit pozycji pionowej końca.
0	SH2	Trzeci bit pozycji poziomej.

! — tylko układy AGA, w poprzednich układach nie używane.

SPRxDATA \$144 W D Rejestr A obrazu danej linii spritea.

SPRxDATB \$146 W D Rejestr B obrazu danej linii spritea.

Ta para rejestrów zawiera dane dla jednej linii spritea. Zazwyczaj są one automatycznie ładowane przez kanały DMA, lecz można je zapisywać przez copper lub Motorolę.

SPRxPTH \$120 W A Adres struktury spritea nr x w pamięci (starsze słowo)

SPRxPTL \$122 W A Adres struktury spritea nr x w pamięci (młodsze słowo)

Ta para rejestrów zawiera adres danych DMA dla spritea nr x (x = 0, 1, 2, 3, 4, 5, 6, 7). Rejestry te muszą być inicjowane przez coppera lub procesor za każdym wygaszaniem pionowym.

STREQU \$038 S D

STRVBL \$03a S D

STRHOR \$03c S DP

STRLONG h \$03e S D

Są to rejestry wykorzystywane przez Amigę w celu zsynchronizowania działania układów komputera z promieniem wizji. Nie należy tych rejestrów zapisywać, ponieważ może to doprowadzić do utraty synchronizacji układów i błędnego działania komputera.

VHPOSR \$006 R A Odczyt pozycji promienia wizji lub pióra świetlnego

VHPOSW \$02c W A Zapis pozycji promienia wizji lub pióra świetlnego

bit#	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
	V7	V6	V5	V4	V3	V2	V1	V0	H8	H7	H6	H5	H4	H3	H2	H1

$$\text{rozdzielczość} = \frac{1}{160 \text{ szerokości ekranu}} \quad (280 \text{ nS})$$

VPOSR p \$004 R Identyfikacja układów (Agnus, Alice) i poz. pionowa promienia wizji (odczyt).

VPOSW \$02a W Identyfikacja układów (Agnus, Alice) i poz. pionowa promienia wizji (zapis).

bit#	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
	LOF	I6	I5	I4	I3	I2	I1	I0	—	—	—	—	—	V10	V9	V8

LOF — long frame (samozmieniający się bit, w BPLCON0),

I0 – I6 — identyfikacja układów:

10 — 8361 (normalny) lub 8370 (Fat, Agnus-ntsc)

00 — 8367 (Pal) lub 8371 (Fat-Pal, Agnus-pal)

20 Pal, 30 NTSC — 8372 (Fat-hr, Agnushr, thru rev 4)

22 Pal, 31 NTSC — 8372 (Fat-hr, Agnushr, rev 5)

22 Pal, 32 NTSC — 8374 (Alice, thru rev 2)

23 Pal, 33 NTSC — 8374 (Alice, rev 3, thru rev 4)

V9, V10 — tylko układy hires (identyfikacja 20, 30)

Dodatek II

Lista rejestrów w kolejności adresowej

Używane symbole:

- & — rejestr używany tylko przez kanały DMA,
- % — rejestr używany zwykle przez kanały DMA, czasem przez procesor,
- +
- ~ — adres nieosiągalny przez coppera gdy bit COPCON = 0,
- h — nowy rejestr dla trybu HIRES,
- p — nowy rejestr układu AGA,
- A — rejestr układu Agnus/Alice,
- D — rejestr układu Denise/Lisa,
- P — rejestr układu Paula,
- W — tylko do zapisu,
- R — tylko do odczytu,
- ER — późny odczyt — jest to rejestr buforujący dane z DMA do RAM-u, z dysku lub z blittera, nie może być odczytywany i zapisywany przez procesor,
- S — zapis tylko zera,

PTL, PTH (pointer) — 20-bitowy adres (wcześniej 18-bitowy) podzielony na dwie części: PTL — młodsze 16 bitów, PTH — wyższe 4 bity. Rejestry te powinny być odświeżane przed użyciem (adresy bitplanów i spriteów co ramkę, a adresy blittera przed jego wystartowaniem) przez procesor lub przez copper. Dzieje się tak dlatego, ponieważ DMA zmienia zawartości tych rejestrów, w celu orientacji co do następnego poboru danych.

LCL, LCH (lokacja) — 20-bitowy adres (wcześniej 18-bitowy) podzielony na dwie części: LCL — młodsze 16 bitów, LCH — wyższe 4 bity. Jest on automatycznie odświeżany (ponieważ jego zawartość zostaje zbuforowana) np. adres copperlisty (odświeżany co ramkę) i długość granego sampla (odświeżony kiedy się skończy).

MOD — 15-bitowe modulo (ze znakiem). Wartość automatycznie dodawana do adresu po wyświetleniu każdej linii. Używane jest też przez blittera do operowania (lub wyświetlania) na oknach mniejszych niż rysunek w pamięci.

<i>nazwa</i>	<i>adres</i>	<i>r/w</i>	<i>układ</i>	<i>opis</i>
BPTDDAT	&~ \$000	ER	A	Rejestru buforujący daną docelowego D dla blittera.
DMACONR	~ \$002	R	AP	Odczyt kontroli DMA i status blittera.

VPOSR	~	\$004	R	A	Identyfikacja układów (Agnus, Alice) i poz. pionowa promienia wizji (zapis).
VHPOSR	~	\$006	R	A	Odczyt pozycji pion.-poziomej promienia wizji.
DSKDATR	&~	\$008	ER	P	Odczyt danej dyskowej.
JOY0DAT	~	\$00a	R	D	Położenie joysticka/myszy w porcie 0.
JOTIDAT	~	\$00c	R	D	Położenie joysticka/myszy w porcie 1.
CLXDAT	~	\$00e	R	D	Rejestr kolizji.
ADKCONR	~	\$010	R	P	Odczyt kontroli dźwięku i dysku.
POT0DAT	~	\$012	R	P	Licznik pot dla portu 0 (paddle, joy. analogowe, itp.)
POT1DAT	~	\$014	R	P	Licznik pot dla portu 1
POTGOR	~	\$016	R	P	Odczyt danych portu dla tzw. pot.
SERDATR	~	\$018	R	P	Odczyt danych i statusu portu szeregowego.
DSKBYTR	~	\$01a	R	P	Odczyt bajtu danej dyskowej i statusu dysku.
INTENAR	~	\$01c	R	P	Odczyt bitów włączenia przerwań.
INTREQR	~	\$01e	R	P	Odczyt bitów zgłoszenia przerwań.
DSKPTH	+~	\$020	W	A	Pointer danych dyskowych (starsze 5 bitów).
DSKPTL	+~	\$022	W	A	Pointer danych dyskowych (młodsze 15 bitów).
DSKLEN	~	\$024	W	P	Długość danych dyskowych w słowach.
DSKDAT	&~	\$026	W	P	Zapis danej dyskowej DMA.
REFPTR	&~	\$028	W	A	Odświeżanie znacznika.
VPOSW	~	\$02a	W	A	Identyfikacja układów (Agnus, Alice) i poz. pionowa promienia wizji (odczyt).
VHPOSW	~	\$02c	W	AD	Zapis poziomej i pionowej pozycji promienia wizji.
COPCON	~	\$02e	W	A	Rejestr kontrolny coppera (CDANG).
SERDAT	~	\$030	W	P	Zapis danej i bitu zatrzymania dla portu szeregowego.
SERPER	~	\$032	W	P	Kontrola portu szeregowego i szybkość transmisji.
POTGO	~	\$034	W	P	Zapis danych portu dla tzw. pot.
JOYTEST	~	\$036	W	D	Zapis wszystkich 4 liczników joysticka/myszy na raz.
STREQU	&~	\$038	S	D	
STRVBL	&~	\$03a	S	D	
STRHOR	&~	\$03c	S	DP	
STRLONG	&~	\$03e	S	D	
BPLCON0	~	\$040	W	A	Rejestr kontrolny blittera nr 0.
BPLCON1	~	\$042	W	A	Rejestr kontrolny blittera nr 1.
BLTAFWM	~	\$044	W	A	Pierwsze słowo maski dla źródła A.
BLTALWM	~	\$046	W	A	Ostatnie słowo maski dla źródła A.
BLTCPTH	+~	\$048	W	A	Adres źródła C (wyższe 5 bitów)
BLTCPTL	+~	\$04a	W	A	Adres źródła C (niższe 15 bitów)

BLTBPTH	+~	\$04c	W	A	Adres źródła B (wyższe 5 bitów)
BLTBPTL	+~	\$04e	W	A	Adres źródła B (niższe 15 bitów)
BLTAPTH	+~	\$050	W	A	Adres źródła A (wyższe 5 bitów)
BLTAPTL	+~	\$052	W	A	Adres źródła A (niższe 15 bitów)
BPTDPPTH	+~	\$054	W	A	Adres przeznaczenia D (wyższe 5 bitów)
BLTDPTL	+~	\$056	W	A	Adres przeznaczenia D (niższe 15 bitów)
BLTSIZE	~	\$058	W	A	Rozmiar operacji blittera i jego start.
BPTCONOL	h~	\$05a	W	A	Kontrola blittera 0. Młodsze 8 bitów (mintermy).
BLTSIZV	h~	\$05c	W	A	Wysokość operacji blittera (15 bitów).
BLTSIZH	h~	\$05e	W	A	Szerokość operacji blittera (11 bitów) i start.
BLTCMOD	~	\$060	W	A	Modulo blittera dla źródła C.
BLTBMOD	~	\$062	W	A	Modulo blittera dla źródła B.
BLTAMOD	~	\$064	W	A	Modulo blittera dla źródła A.
BLTDMOD	~	\$066	W	A	Modulo blittera dla przeznaczenia D.
	~	\$068			
	~	\$06a			
	~	\$06c			
	~	\$06e			
BLTCDAT	&~	\$070	W	A	Rejestr danych źródła C.
BLTBDAT	&~	\$072	W	A	Rejestr danych źródła B.
BLTADAT	&~	\$074	W	A	Rejestr danych źródła A.
	~	\$076			
SPRH DAT	&h~	\$078	W	A	Rozszerzony logiczny UHRES pointer spritea i identyfikator danych.
(BPLH DAT)	~	\$07a			
LISAID	h~	\$07c	R	D	Wersja układu Denise/Lisa.
DSKSYNC	~	\$07e	W	P	Wzorzec synchronizacji dla danej dyskowej.
COP1LCH	+	\$080	W	A	Adres pierwszej lokacji coppera (starsze 5 bitów)
COP1LCL	+	\$082	W	A	Adres pierwszej lokacji coppera (młodsze 15 bitów)
COP2LCH	+	\$084	W	A	Adres drugiej lokacji coppera (starsze 5 bitów)
COP2LCL	+	\$086	W	A	Adres drugiej lokacji coppera (młodsze 15 bitów)
COPJMP1		\$088	S	A	Restart coppera od pierwszej lokacji.
COPJMP2		\$08a	S	A	Restart coppera od drugiej lokacji.
COPINS		\$08c	W	A	Identyfikacja pobranej instrukcji coppera.
DIWSTR T		\$08e	W	AD	Pozycja początku okna wyświetlania.
DIWSTOP		\$090	W	AD	Pozycja końca okna wyświetlania.
DDFSTR T		\$092	W	A	Pozycja pozioma początku wyświetlania danych.
DDFSTOP		\$094	W	A	Pozycja pozioma końca wyświetlania danych.
DMA CON		\$096	W	AP	Zapis kontroli DMA.
CLX CON		\$098	W	D	Kontrola kolizji.

INTENA		\$09a	W	P	Bity włączenia przerwań.
INTREQ		\$09c	W	P	Bity zgłoszenia przerwań.
ADKCON		\$09e	W	P	Kontrola dźwięku, dysku i UART.
AUD0LCH	+	\$0a0	W	A	Lokacja danych dźwiękowych kanału 0 (starsze 5 bitów).
AUD0LCL	+	\$0a2	W	A	Lokacja danych dźwiękowych kanału 0 (młodsze 15 bitów)
AUD0LEN		\$0a4	W	P	Długość danych kanału 0.
AUD0PER		\$0a6	W	P	Okres kanału 0.
AUD0VOL		\$0a8	W	P	Głośność kanału 0.
AUD0DAT	&	\$0aa	W	P	Dana dla kanału dźwiękowego 0.
		\$0ac			
		\$0ae			
AUD1LCH	+	\$0b0	W	A	Lokacja danych dźwiękowych kanału 1 (starsze 5 bitów).
AUD1LCL	+	\$0b2	W	A	Lokacja danych dźwiękowych kanału 1 (młodsze 15 bitów)
AUD1LEN		\$0b4	W	P	Długość danych kanału 1.
AUD1PER		\$0b6	W	P	Okres kanału 1.
AUD1VOL		\$0b8	W	P	Głośność kanału 1.
AUD1DAT	&	\$0ba	W	P	Dana dla kanału dźwiękowego 1.
		\$0bc			
		\$0be			
AUD2LCH	+	\$0c0	W	A	Lokacja danych dźwiękowych kanału 2 (starsze 5 bitów).
AUD2LCL	+	\$0c2	W	A	Lokacja danych dźwiękowych kanału 2 (młodsze 15 bitów)
AUD2LEN		\$0c4	W	P	Długość danych kanału 2.
AUD2PER		\$0c6	W	P	Okres kanału 2.
AUD2VOL		\$0c8	W	P	Głośność kanału 2.
AUD2DAT	&	\$0ca	W	P	Dana dla kanału dźwiękowego 2.
		\$0cc			
		\$0ce			
AUD3LCH	+	\$0d0	W	A	Lokacja danych dźwiękowych kanału 3 (starsze 5 bitów).
AUD3LCL	+	\$0d2	W	A	Lokacja danych dźwiękowych kanału 3 (młodsze 15 bitów)
AUD3LEN		\$0d4	W	P	Długość danych kanału 3.
AUD3PER		\$0d6	W	P	Okres kanału 3.
AUD3VOL		\$0d8	W	P	Głośność kanału 3.

AUD3DAT	&	\$0da	W	P	Dana dla kanału dźwiękowego 3.
		\$0dc			
		\$0de			
BPL1PTH	+	\$0e0	W	A	Pointer bitplanu 1 (wyższe 5 bitów)
BPL1PTL	+	\$0e2	W	A	Pointer bitplanu 1 (niższe 15 bitów)
BPL2PTH	+	\$0e4	W	A	Pointer bitplanu 2 (wyższe 5 bitów)
BPL2PTL	+	\$0e6	W	A	Pointer bitplanu 2 (niższe 15 bitów)
BPL3PTH	+	\$0e8	W	A	Pointer bitplanu 3 (wyższe 5 bitów)
BPL3PTL	+	\$0ea	W	A	Pointer bitplanu 3 (niższe 15 bitów)
BPL4PTH	+	\$0ec	W	A	Pointer bitplanu 4 (wyższe 5 bitów)
BPL4PTL	+	\$0ee	W	A	Pointer bitplanu 4 (niższe 15 bitów)
BPL5PTH	+	\$0f0	W	A	Pointer bitplanu 5 (wyższe 5 bitów)
BPL5PTL	+	\$0f2	W	A	Pointer bitplanu 5 (niższe 15 bitów)
BPL6PTH	+	\$0f4	W	A	Pointer bitplanu 6 (wyższe 5 bitów)
BPL6PTL	+	\$0f6	W	A	Pointer bitplanu 6 (niższe 15 bitów)
BPL7PTH	+p	\$0f8	W	A	Pointer bitplanu 7 (wyższe 5 bitów)
BPL7PTL	+p	\$0fa	W	A	Pointer bitplanu 7 (niższe 15 bitów)
BPL8PTH	+p	\$0fc	W	A	Pointer bitplanu 8 (wyższe 5 bitów)
BPL8PTL	+p	\$0fe	W	A	Pointer bitplanu 8 (niższe 15 bitów)
BPLCON0		\$100	W	AD	Rejestr kontrolny bitplanów nr 0
BPLCON1		\$102	W	D	Rejestr kontrolny bitplanów nr 1
BPLCON2		\$104	W	D	Rejestr kontrolny bitplanów nr 2
BPLCON3	p	\$106	W	D	Rejestr kontrolny bitplanów nr 3
BPL1MOD		\$108	W	A	Modulo bitplanów nieparzystych.
BPL2MOD		\$10a	W	A	Modulo bitplanów parzystych.
BPLCON4	p	\$10c	W	D	Rejestr kontrolny bitplanów nr 4
CLXCON2	p	\$10e	W	D	Rejestr rozszerzonej kontroli kolizji
BPL1DAT	&	\$110	W	D	Dana do wyświetlania bitplanu 1.
BPL2DAT	&	\$112	W	D	Dana do wyświetlania bitplanu 2.
BPL3DAT	&	\$114	W	D	Dana do wyświetlania bitplanu 3.
BPL4DAT	&	\$116	W	D	Dana do wyświetlania bitplanu 4.
BPL5DAT	&	\$118	W	D	Dana do wyświetlania bitplanu 5.
BPL6DAT	&	\$11a	W	D	Dana do wyświetlania bitplanu 6.
BPL7DAT	&p	\$11c	W	D	Dana do wyświetlania bitplanu 7.
BPL8DAT	&p	\$11e	W	D	Dana do wyświetlania bitplanu 8.
SPR0PTH	+	\$120	W	A	Pointer spritea nr 0 (wyższe 5 bitów)
SPR0PTL	+	\$122	W	A	Pointer spritea nr 0 (niższe 15 bitów)
SPR1PTH	+	\$124	W	A	Pointer spritea nr 1 (wyższe 5 bitów)
SPR1PTL	+	\$126	W	A	Pointer spritea nr 1 (niższe 15 bitów)
SPR2PTH	+	\$128	W	A	Pointer spritea nr 2 (wyższe 5 bitów)

SPR2PTL	+	\$12a	W	A	Pointer spritea nr 2 (niższe 15 bitów)
SPR3PTH	+	\$12c	W	A	Pointer spritea nr 3 (wyższe 5 bitów)
SPR3PTL	+	\$12e	W	A	Pointer spritea nr 3 (niższe 15 bitów)
SPR4PTH	+	\$130	W	A	Pointer spritea nr 4 (wyższe 5 bitów)
SPR4PTL	+	\$132	W	A	Pointer spritea nr 4 (niższe 15 bitów)
SPR5PTH	+	\$134	W	A	Pointer spritea nr 5 (wyższe 5 bitów)
SPR5PTL	+	\$136	W	A	Pointer spritea nr 5 (niższe 15 bitów)
SPR6PTH	+	\$138	W	A	Pointer spritea nr 6 (wyższe 5 bitów)
SPR6PTL	+	\$13a	W	A	Pointer spritea nr 6 (niższe 15 bitów)
SPR7PTH	+	\$13c	W	A	Pointer spritea nr 7 (wyższe 5 bitów)
SPR7PTL	+	\$13e	W	A	Pointer spritea nr 7 (niższe 15 bitów)
SPR0POS	%	\$140	W	AD	Pozycja pion.-poz. spritea 0; start
SPR0CTL	%	\$142	W	AD	Pozycja pionowa końca spritea 0
SPR0DATA	%	\$144	W	D	Rejestr A danej obrazu spritea 0
SPR0DATB	%	\$146	W	D	Rejestr B danej obrazu spritea 0
SPR1POS	%	\$148	W	AD	Pozycja pion.-poz. spritea 1; start
SPR1CTL	%	\$14a	W	AD	Pozycja pionowa końca spritea 1
SPR1DATA	%	\$14c	W	D	Rejestr A danej obrazu spritea 1
SPR1DATB	%	\$14e	W	D	Rejestr B danej obrazu spritea 1
SPR2POS	%	\$150	W	AD	Pozycja pion.-poz. spritea 2; start
SPR2CTL	%	\$152	W	AD	Pozycja pionowa końca spritea 2
SPR2DATA	%	\$154	W	D	Rejestr A danej obrazu spritea 2
SPR2DATB	%	\$156	W	D	Rejestr B danej obrazu spritea 2
SPR3POS	%	\$158	W	AD	Pozycja pion.-poz. spritea 3; start
SPR3CTL	%	\$15a	W	AD	Pozycja pionowa końca spritea 3
SPR3DATA	%	\$15c	W	D	Rejestr A danej obrazu spritea 3
SPR3DATB	%	\$15e	W	D	Rejestr B danej obrazu spritea 3
SPR4POS	%	\$160	W	AD	Pozycja pion.-poz. spritea 4; start
SPR4CTL	%	\$162	W	AD	Pozycja pionowa końca spritea 4
SPR4DATA	%	\$164	W	D	Rejestr A danej obrazu spritea 4
SPR4DATB	%	\$166	W	D	Rejestr B danej obrazu spritea 4
SPR5POS	%	\$168	W	AD	Pozycja pion.-poz. spritea 5; start
SPR5CTL	%	\$16a	W	AD	Pozycja pionowa końca spritea 5
SPR5DATA	%	\$16c	W	D	Rejestr A danej obrazu spritea 5
SPR5DATB	%	\$16e	W	D	Rejestr B danej obrazu spritea 5
SPR6POS	%	\$170	W	AD	Pozycja pion.-poz. spritea 6; start
SPR6CTL	%	\$172	W	AD	Pozycja pionowa końca spritea 6
SPR6DATA	%	\$174	W	D	Rejestr A danej obrazu spritea 6
SPR6DATB	%	\$176	W	D	Rejestr B danej obrazu spritea 6
SPR7POS	%	\$178	W	AD	Pozycja pion.-poz. spritea 7; start

SPR7CTL	%	\$17a	W	AD	Pozycja pionowa końca spritea 7
SPR7DATA	%	\$17c	W	D	Rejestr A danej obrazu spritea 7
SPR7DATB	%	\$17e	W	D	Rejestr B danej obrazu spritea 7
COLOR00	f	\$180	W	D	Rejestr koloru nr 00
COLOR01	f	\$182	W	D	Rejestr koloru nr 01
COLOR02	f	\$184	W	D	Rejestr koloru nr 02
COLOR03	f	\$186	W	D	Rejestr koloru nr 03
COLOR04	f	\$188	W	D	Rejestr koloru nr 04
COLOR05	f	\$18a	W	D	Rejestr koloru nr 05
COLOR06	f	\$18c	W	D	Rejestr koloru nr 06
COLOR07	f	\$18e	W	D	Rejestr koloru nr 07
COLOR08	f	\$190	W	D	Rejestr koloru nr 08
COLOR09	f	\$192	W	D	Rejestr koloru nr 09
COLOR10	f	\$194	W	D	Rejestr koloru nr 10
COLOR11	f	\$196	W	D	Rejestr koloru nr 11
COLOR12	f	\$198	W	D	Rejestr koloru nr 12
COLOR13	f	\$19a	W	D	Rejestr koloru nr 13
COLOR14	f	\$19c	W	D	Rejestr koloru nr 14
COLOR15	f	\$19e	W	D	Rejestr koloru nr 15
COLOR16	f	\$1a0	W	D	Rejestr koloru nr 16
COLOR17	f	\$1a2	W	D	Rejestr koloru nr 17
COLOR18	f	\$1a4	W	D	Rejestr koloru nr 18
COLOR19	f	\$1a6	W	D	Rejestr koloru nr 19
COLOR20	f	\$1a8	W	D	Rejestr koloru nr 20
COLOR21	f	\$1aa	W	D	Rejestr koloru nr 21
COLOR22	f	\$1ac	W	D	Rejestr koloru nr 22
COLOR23	f	\$1ae	W	D	Rejestr koloru nr 23
COLOR24	f	\$1b0	W	D	Rejestr koloru nr 24
COLOR25	f	\$1b2	W	D	Rejestr koloru nr 25
COLOR26	f	\$1b4	W	D	Rejestr koloru nr 26
COLOR27	f	\$1b6	W	D	Rejestr koloru nr 27
COLOR28	f	\$1b8	W	D	Rejestr koloru nr 28
COLOR29	f	\$1ba	W	D	Rejestr koloru nr 29
COLOR30	f	\$1bc	W	D	Rejestr koloru nr 30
COLOR31	f	\$1be	W	D	Rejestr koloru nr 31

Poniżej znajdują się rejestry AGA lub ECS

HTOTAL	h	\$1c0	W	A	Największy numer licznika poziomej linii (VARBEAMEN=1)
HSSTOP	h	\$1c2	W	A	Pozioma pozycja linii dla zakończenia HSYNC.

HBSTRT	h	\$1c4	W	AD	Pozioma pozycja linii dla rozpoczęcia wygaszania poziomego (HBLANK)
HBSTOP	h	\$1c6	W	AD	Pozioma pozycja linii dla zakończenia wygaszania poziomego (HBLANK)
VTOTAL	h	\$1c8	W	A	Największy numer licznika pionowej linii (VARBEAMEN=1)
VSSTOP	h	\$1ca	W	A	Pionowa pozycja linii dla zakończenia VSYNC.
VBSTRT	h	\$1cc	W	A	Pionowa pozycja linii dla rozpoczęcia wygaszania pionowego (VBLANK)
VBSTOP	h	\$1ce	W	A	Pionowa pozycja linii dla zakończenia wygaszania pionowego (VBLANK)
SPRHSTRT	h	\$1d0	W	A	Pozycja pionowa początku spritea UHRES.
SPRHSTOP	h	\$1d2	W	A	Pozycja pionowa końca spritea UHRES.
BPLHSTRT	h	\$1d4	W	A	Pozycja pionowa początku bitplanu UHRES.
BPLHSTOP	h	\$1d6	W	A	Pozycja pionowa końca bitplanu UHRES.
HHPOSW	h	\$1d8	W	A	Zapis pozycji poziomej promienia dla podwójnego trybu hires
HHPOSR	h	\$1da	R	A	Odczyt pozycji poziomej promienia dla podwójnego trybu hires
BEAMCON0	h	\$1dc	W	A	Rejestr kontrolny licznika promienia (SHRES, UHRES, PAL)
HSSTRT	h	\$1de	W	A	Pozycja pozioma początku synchronizacji (VARHSY)
VSSTRT	h	\$1e0	W	A	Pozycja pionowa początku synchronizacji (VARHSY)
HCENTER	h	\$1e2	W	A	Pozycja pozioma dla synchronizacji pionowej w trybie interlace
DIWHIGH	h	\$1e4	W	AD	Starsze bity pozycji początku i końca okna wyświetlania
BPLHMOD	h	\$1e6	W	A	Modulo bitplanu UHRES
SPRHPTH	+h	\$1e8	W	A	Pointer spritea UHRES (wyższe 5 bitów)
SPRHPTL	+h	\$1ea	W	A	Pointer spritea UHRES (niższe 15 bitów)
BPLHPH	+h	\$1ec	W	A	VRam (UHRES) Pointer bitplanu (wyższe 5 bitów)
BPLHPTL	+h	\$1ee	W	A	VRam (UHRES) Pointer bitplanu (niższe 15 bitów)
RESERVED	\$1f0-\$1fa				
FMODE	p	\$1fc	W	AD	Rejestr kontrolujący pobieranie danych.
NO-OP		\$1fe			
NULL					

Uwaga: aby uzyskać więcej informacji zajrzyj do alfabetycznego spisu rejestrów.

Dodatek III

Omówienie pozostałych rozkazów Motoroli 68000

Rozkaz STOP (Load status register and stop — załadowanie rejestru statusowego i zatrzymanie)

Składnia:

STOP #<dana>

Atrybuty:

brak

Działanie:

Instrukcja STOP służy do jednoczesnego zezwolenia na przerwanie i do wprowadzenia procesora w stan oczekiwania na przerwanie.

Instrukcja ta jest instrukcją uprzywilejowaną. Zawartość 16-bitowej danej jest ładowana do rejestru statusowego. Bit 13 tej danej (odpowiadający bitowi nadzorcy S) musi być ustawiony.

Kody warunków:

ustawiane zgodnie z daną natychmiastową zawartą w instrukcji

Dozwolone tryby adresowania:

żaden

Rozkaz RESET (Reset external devices — zerowanie urządzeń zewnętrznych)

Składnia:

RESET

Atrybuty:

brak

Działanie:

Instrukcja RESET powoduje wyzerowanie wszystkich urządzeń zewnętrznych. Jest instrukcją uprzywilejowaną.

Kody warunków:

- X — nie zmieniany,
- N — nie zmieniany,
- Z — nie zmieniany,
- V — nie zmieniany,
- C — nie zmieniany.

Dozwolone tryby adresowania:

żaden

Rozkaz Sxx (Set according to condition — ustawianie w zależności od warunku)**Składnia:**

Sxx <ea>

Atrybuty:

rozmiar: B

Działanie:

Instrukcja powoduje ustawienie bajtu określonego adresem efektywnym na wartość \$ff, gdy spełniony zostanie określony warunek. Bajt zostaje wyzerowany, gdy warunek nie zostaje spełniony.

Dopuszczalne są następujące instrukcje:

- SCC — bajt zostaje ustawiony, gdy bit C jest wyzerowany,
- SCS — bajt zostaje ustawiony, gdy bit C jest ustawiony,
- SEQ — ustawienie, gdy równy (Z = 1),
- SGE — ustawienie, gdy większy lub równy (N = 0, V = 1 lub N = 0, V = 0),
- SGT — ustawienie, gdy większy (N = 1, V = 1, Z = 0 lub N = 0, V = 0, Z = 0),
- SHI — ustawienie, gdy wyższy (C = 0, Z = 0), stosowane przy liczbach bez znaku,
- SLE — ustawienie, gdy mniejszy lub równy (Z = 1 lub N = 1, V = 0 lub N = 0, V = 1),
- SLS — ustawienie, gdy niższy lub taki sam (C = 1 lub Z = 1), stosowane przy liczbach bez znaku,
- SLT — ustawienie, gdy mniejszy (N = 1, V = 0 lub N = 0, V = 1),
- SMI — ustawienie, gdy ujemny (N = 1),
- SNE — ustawienie, gdy różny (Z = 0),
- SPL — ustawienie, gdy dodatni (N = 0),
- SVC — ustawienie, gdy V = 0,
- SVS — ustawienie, gdy V = 1,
- SF — bajt nigdy nie zostanie ustawiony,
- ST — bajt zawsze zostanie ustawiony.

Kody warunków:

X — nie zmieniany,
 N — nie zmieniany,
 Z — nie zmieniany,
 V — nie zmieniany,
 C — nie zmieniany.

Dozwolone tryby adresowania:

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
tak	nie	tak	tak	tak	tak	tak
X.w	X.1	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	nie	nie	nie	nie	nie

Przykład:

```

CMP.W    #320,D0
SGE      D1
CMP.W    #0,D0
SLT      D2
OR.B     D1,D2
BNE      Poza_Granicą
...
Poza_Granicą:
...
```

Powyższy przykład testuje, czy zawartość młodsze słowa rejestru D0 jest pomiędzy 0 a 319. Jeśli tak nie jest, to procesor skacze do etykiety Poza_Granicą. Metoda ta jest znacznie szybsza od metody wykorzystującej instrukcje Bxx.

Rozkaz NBCD (Negate decimal with extend
— negowanie dziesiętne z rozszerzeniem)

Składnia:

NBCD <ea>

Atrybuty:

rozmiar: B

Działanie:

Instrukcja NBCD przeprowadza operację negowania liczby zapisane w kodzie BCD. Wykorzystywaną techniką jest uzupełnienie do dziesięciu, które jest analogiczne z uzupełnieniem do dwóch dla liczb binarnych. Uzupełnienie do dziesięciu polega na odjęciu danej liczby od liczby składającej się z samych dziewiątek i dodaniu do wyniku jedynki. Podobnie jak dla liczb binarnych, proces uzupełnienia pracuje poprawnie jedynie, gdy operację wykonuje się na określone liczbie cyfr.

Kody warunków:

- X — Ustawiany gdy wystąpi pożyczka w najbardziej znaczącej cyfrze BCD, w przeciwnym wypadku kasowany,
 N — niezdefiniowany,
 Z — ustawiany gdy wynik operacji jest zerowy, w przeciwnym wypadku zerowany,
 V — niezdefiniowany,
 C — ustawiany gdy wystąpi pożyczka w najbardziej znaczącej cyfrze BCD, w przeciwnym wypadku kasowany.

Dozwolone tryby adresowania:

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
tak	nie	nie	nie	tak	nie	nie
X.w	X.1	x(Pc)	x(Pc,R.s)	#x	SR	CCR
nie	nie	nie	nie	nie	nie	nie

Przykład:

NECD -(A6)

Bajt znajdujący się pod adresem wskazywanym przez rejestr A6 zostanie zanegowany w kodzie BCD, a zawartość rejestru A6 zmniejszy się o 1.

Rozkaz SBCD (Subtract decimal with extend

— *odejmowanie dziesiętne z rozszerzeniem*)

Składnia:

SBCD Dy,Dx

SBCD -(Ay),-(Ax)

Atrybuty:

rozmiar: B

Działanie:

Instrukcja SBCD odejmuje operand źródłowy od operandu docelowego z uwzględnieniem bitu rozszerzenia X. Odejmowanie odbywa się w kodzie BCD. Wynik operacji zostaje zapisany w miejscu przeznaczenia. Istnieją dwie formy tej instrukcji:

- odejmowany jest rejestr danych od drugiego rejestru danych,
- pamięć od pamięci: operandy są określane przez rejestry adresowe, adresowane w trybie predekrement.

Instrukcja SBCD ma rozmiar bajtu.

Kody warunków:

- X — ustawiany gdy wystąpi pożyczka w najbardziej znaczącej cyfrze BCD, w przeciwnym wypadku kasowany,
- N — niezdefiniowany,
- Z — ustawiany gdy wynik operacji jest zerowy, w przeciwnym wypadku zerowany,
- V — niezdefiniowany,
- C — ustawiany gdy wystąpi pożyczka w najbardziej znaczącej cyfrze BCD, w przeciwnym wypadku kasowany.

Dozwolone tryby adresowania:

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
tak	nie	nie	nie	tak	nie	nie
X.w	X.l	x(Pc)	x(Pc,R.s)	#x	SR	CCR
nie	nie	nie	nie	nie	nie	nie

Przykład:

SBCD D0,D1

Wynikiem tych dwóch rozkazów będzie dana, o długości bajtu, zapisana w kodzie BCD, będąca różnicą najmłodszych bajtów rejestru D0 i D1.

Rozkaz ABCD (Add decimal with extend
— dodawanie dziesiętne z rozszerzeniem)

Składnia:

ABCD Dy,Dx

ABCD -(Ay),-(Ax)

Atrybuty:

rozmiar: B

Działanie:

Instrukcja ABCD dodaje operand źródłowy do operandu docelowego z uwzględnieniem bitu rozszerzenia X. Dodawanie odbywa się w kodzie BCD. Wynik operacji zostaje zapisany w miejscu przeznaczenia. Istnieją dwie formy tej instrukcji:

- dodawany jest rejestr danych do drugiego rejestru danych,
- pamięć do pamięci: operandy są określane przez rejestry adresowe, adresowane w trybie predekrement.

Instrukcja ABCD ma rozmiar bajtu.

Kody warunków:

- X — ustawiany gdy wystąpi przeniesienie (dziesiętne), w przeciwnym wypadku kasowany,
 N — niezdefiniowany,
 Z — ustawiany gdy wynik operacji jest zerowy, w przeciwnym wypadku zerowany,
 V — niezdefiniowany,
 C — ustawiany gdy wystąpi przeniesienie (dziesiętne), w przeciwnym wypadku kasowany.

Dozwolone tryby adresowania:

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
tak	nie	nie	nie	tak	nie	nie
X.w	X.1	x(Pc)	x(Pc,R.s)	#x	SR	CCR
nie	nie	nie	nie	nie	nie	nie

Przykład:

ABCD - (A0), -(A1)
 ABCD - (A0), -(A1)

Wynikiem tych dwóch rozkazów będzie dana o długości słowa, zapisana w kodzie BCD.

Rozkaz NEGX (Negate with extend — negacja z rozszerzeniem)**Składnia:**

NEGX <ea>

Atrybuty:

rozmiar: B, W, L

Działanie:

Instrukcja NEGX (porównaj NEG) umożliwia negowanie liczb większych niż 32-bitowe. Wykorzystuje w tym celu bit X rejestru systemowego (porównaj ADDX, SUBX).

Kody warunków:

- X — ustawiany gdy wygenerowana została pożyczka, w przeciwnym wypadku kasowany,
 N — ustawiany, gdy wynik jest ujemny, w przeciwnym wypadku zerowany,
 Z — zerowany, gdy wynik jest różny od zera, w przeciwnym wypadku ustawiany,
 V — ustawiany, gdy zgłoszony został nadmiar, w przeciwnym wypadku zerowany,
 C — ustawiany gdy wygenerowana została pożyczka, w przeciwnym wypadku kasowany.

Dozwolone tryby adresowania:

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
tak	tak	tak	tak	tak	tak	tak
X.w	X.1	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	nie	nie	nie	nie	nie

Przykład:

NEG.L - (A0)

NEGX.L - (A0)

NEGX.L - (A0)

Powyższy przykład neguje 96-bitową liczbę zapisaną pod adresem wskazywanym przez zawartość rejestru A0.

Rozkaz TAS (Test and set an operand — testowanie i ustawienie operandu)**Składnia:**

TAS <ea>

Atrybuty:

rozmiar: B

Działanie:

Instrukcja TAS testuje określony adresem efektywnym bajt. Najstarszy bit testowanego bajtu zostaje ustawiony na 1. Bity N i Z zostają ustawione, zgodnie z wartością bajtu przed wykonaniem operacji.

Kody warunków:

X — nie zmieniany,

N — ustawiany, gdy najstarszy bit operandu równy jest jeden, przed operacją, w przeciwnym wypadku kasowany,

Z — ustawiany, gdy wszystkie bity operandu były wyzerowane przed operacją, w przeciwnym wypadku kasowany,

V — nie zmieniany,

C — nie zmieniany.

Dozwolone tryby adresowania:

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
tak	nie	tak	tak	tak	tak	tak
X.w	X.1	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	nie	nie	nie	nie	nie

Rozkaz CMPA (Compare address — porównanie adresu)**Składnia:**

CMPA <ea>,An

Atrybuty:

rozmiar: W, L

Działanie:

Działanie instrukcji CMPA polega na porównaniu zawartości rejestru adresowego z operandem określonym adresem efektywnym. Porównanie to polega na odjęciu od rejestru adresowego operandu źródłowego i, w zależności od wyniku, odpowiedniego ustawienia kodów warunków. Żaden z operandów nie pozostaje zmieniony. Rozmiar instrukcji może być określony jako słowo lub długie słowo.

Kody warunków:

- X — nie zmieniany,
- N — ustawiany gdy wynik jest ujemny, w przeciwnym wypadku zerowany,
- Z — ustawiany gdy wynik jest zerowy, w przeciwnym wypadku zerowany,
- V — ustawiany gdy wystąpił nadmiar przy odejmowaniu, w przeciwnym wypadku zerowany,
- C — ustawiany gdy wygenerowana została pożyczka, w przeciwnym wypadku zerowany.

Dozwolone tryby adresowania:

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
tak	tak	tak	tak	tak	tak	tak
X.w	X.l	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	tak	tak	tak	nie	nie

Przykład:

```

Loop:  MOVE.B  #0,(A0)+
        CMPA.L  A0,A1
        BNE    Loop

```

Przykład ten czyści pamięć od adresu wskazywanego przez A0 do adresu wskazywanego przez A1. Należy jednak uważać, by A0 było mniejsze od A1.

Rozkaz CLR (Clear an operand — zerowanie operandu)**Składnia:**

CLR <ea>

Atrybuty:

rozmiar: B, W, L

Działanie:

Instrukcja CLR powoduje wyzerowanie operandu przeznaczenia. Rozmiar instrukcji może być określony jako bajt, słowo lub długie słowo.

Kody warunków:

X — nie zmieniany,
 N — zawsze zerowany,
 Z — zawsze zerowany,
 V — zawsze zerowany,
 C — zawsze zerowany.

Dozwolone tryby adresowania:

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R,s)
tak	nie	tak	tak	tak	tak	tak
X.w	X.l	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	nie	nie	nie	nie	nie

Rozkaz CMPM (Compare memory — porównanie pamięci)**Składnia:**

CMPM (Ay)+,(Ax)+

Atrybuty:

rozmiar: B, W, L

Działanie:

Instrukcja CMPM porównuje dwa operandy pamięciowe, z użyciem trybu adresowania postinkrement. Po odjęciu operandu źródłowego od operandu przeznaczenia, odpowiednio ustawiane są kody warunków. Żaden z operandów nie zostaje zmieniony. Rozmiar instrukcji może być określony jako bajt, słowo lub długie słowo.

Kody warunków:

X — nie zmieniany,
 N — ustawiany gdy wynik jest ujemny, w przeciwnym wypadku zerowany,
 Z — ustawiany gdy wynik jest zerowy, w przeciwnym wypadku zerowany,
 V — ustawiany gdy wystąpił nadmiar przy odejmowaniu, w przeciwnym wypadku zerowany,
 C — ustawiany gdy wygenerowana została pożyczka, w przeciwnym wypadku zerowany.

Dozwolone tryby adresowania:

jedynie tryb adresowania postinkrement

Przykład:

```

      MOVEQ    #512-1,D7
Loop:  CPM.B    (A0)+,(A1)+
      BEQ      Ok
      BRA      No_Ok
Ok:    DBF      D7,Loop
      BRA      All_Ok
No_Ok:  MOVEQ    #0,D0
      RTS
All_Ok:  MOVEQ    #1,D0
      RTS

```

Zadaniem powyższej procedury jest porównanie dwóch 512 bajtowych ciągów danych umieszczonych pod adresami wskazywanymi przez rejestry A0 i A1. W wypadku różnicy tych ciągów, procedura zwróci wartość 0.

Rozkaz ADDX (Add extended — dodawanie z rozszerzeniem)**Składnia:**

```

ADDX  Dx,Dy
ADDX  -(Ay),-(Ax)

```

Atrybuty:

rozmiar: B, W, L

Działanie:

Instrukcja ADDX dodaje operand źródłowy do operandu przeznaczenia, łącznie z bitem rozszerzenia X. Wynik dodawania zapisywany jest w miejscu przeznaczenia. Istnieją dwie formy tej instrukcji:

- dodawany jest rejestr danych do drugiego rejestru danych,
- pamięć do pamięci: operandy są określane przez rejestry adresowe, adresowane w trybie predekrement.

Instrukcja ADDX służy do dodawania liczb większych niż 32-bitowe.

Kody warunków:

- X — ustawiany gdy wystąpi przeniesienie, w przeciwnym wypadku kasowany,
- N — ustawiany wtedy, gdy najstarszy bit wyniku równy jest jeden, w przeciwnym wypadku jest zerowany,
- Z — ustawiany gdy wynik operacji jest zerowy, w przeciwnym wypadku zerowany,
- V — ustawiany, gdy wystąpi nadmiar, w przeciwnym wypadku zerowany,
- C — ustawiany gdy wystąpi przeniesienie, w przeciwnym wypadku kasowany.

Dozwolone tryby adresowania:

Dn	An	(An)	(An)+	-(An)	x(An)	x(An.R.s)
tak	nie	nie	nie	tak	nie	nie
X.w	X.l	x(Pc)	x(Pc,R.s)	#x	SR	CCR
nie	nie	nie	nie	nie	nie	nie

Przykład:

ADD.L D2,D4

ADDX.L D1,D3

Wynikiem tego dodawania będzie 64-bitowa liczba w rejestrach D3 i D4.

Rozkaz SUBX (Subtract extended — odejmowanie z rozszerzeniem)**Składnia:**

SUBX Dx,Dy

SUBX -(Ay),-(Ax)

Atrybuty:

rozmiar: B, W, L

Działanie:

Instrukcja SUBX odejmuje operand źródłowy od operandu przeznaczenia, łącznie z bitem rozszerzenia X. Wynik odejmowania zapisywany jest w miejscu przeznaczenia. Istnieją dwie formy tej instrukcji:

- odejmowany jest rejestr danych od drugiego rejestru danych,
- pamięć od pamięci: operandy są określone przez rejestry adresowe, adresowane w trybie predekrement.

Instrukcja SUBX służy do odejmowania liczb większych niż 32-bitowe.

Kody warunków:

- X — ustawiany gdy wystąpi pożyczka z najstarszego bitu, w przeciwnym wypadku kasowany,
- N — ustawiany gdy wynik operacji jest ujemny, w przeciwnym wypadku jest zerowany,
- Z — zerowany gdy wynik operacji jest różny od zera, w przeciwnym wypadku nie jest zmieniany,
- V — ustawiany, gdy wystąpi nadmiar, w przeciwnym wypadku zerowany,
- C — ustawiany gdy wystąpi pożyczka z najstarszego bitu, w przeciwnym wypadku kasowany.

Dozwolone tryby adresowania:

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
tak	nie	nie	nie	tak	nie	nie
X.w	X.l	x(Pc)	x(Pc,R.s)	#x	SR	CCR
nie	nie	nie	nie	nie	nie	nie

Przykład:

SUBX.L (A1)-(A0)-
 SUBX.L (A1),(A0)

Wynikiem tego odejmowania będzie 64-bitowa liczba pod adresem wskazywanym przez zawartość rejestru A0.

Rozkaz PEA (Push effective address — odłożenie adresu efektywnego na stos)**Składnia:**

PEA <ea>

Atrybuty:

rozmiar: L

Działanie:

Instrukcja PEA umieszcza na wierzchołku stosu obliczony adres efektywny. Rozmiar operacji ograniczony jest do długiego słowa.

Kody warunków:

- X — nie zmieniany,
- N — nie zmieniany,
- Z — nie zmieniany,
- V — nie zmieniany,
- C — nie zmieniany.

Dozwolone tryby adresowania:

Dn	An	(An)	(An)+	-(An)	x(An)	x(An,R.s)
nie	nie	tak	nie	nie	tak	tak
X.w	X.l	x(Pc)	x(Pc,R.s)	#x	SR	CCR
tak	tak	tak	tak	nie	nie	nie

Przykład:

```
PEA      6 (A4, D0.w)
JMP      (A7)
```

Na wierzchołku stosu zostanie umieszczony adres będący sumą zawartości rejestrów: A4, młodsze słowa rejestru D0 i liczby 6. Następnie nastąpi skok do obliczonego adresu.

Rozkaz LINK (Link and allocate — łączenie i alokacja)**Składnia:**

LINK An, #<przesunięcie>

Atrybuty:

bez rozmiaru

Działanie:

Instrukcja LINK rezerwuje roboczy obszar pamięci na stosie. Pobiera ona dwa operandy: rejestr adresowy oraz 24-bitowe przesunięcie ze znakiem (czyli może być ono ujemne). Zawartość rejestru adresowego odkładana jest na stosie, po czym zawartość licznika stosu zostaje do niego skopiowana. Przesunięcie (zazwyczaj ujemne), dodane zostaje do wskaźnika stosu.

Kody warunków:

X — nie zmieniany,
N — nie zmieniany,
Z — nie zmieniany,
V — nie zmieniany,
C — nie zmieniany.

Dozwolone tryby adresowania:

żaden.

Przykład:

```
LINK     A2, #-200
```

Na programowym stosie wskazywanym przez rejestr A2, zostanie zarezerwowana przestrzeń 200 bajtów.

Rozkaz UNLK (Unlink — zwolnienie zarezerwowanego obszaru)

UNLK An

Atrybuty:

brak

Działanie:

Instrukcja UNLK zwalnia zarezerwowany wcześniej instrukcją LINK roboczy obszar pamięci na stosie.

Kody warunków:

- X — nie zmieniany,
- N — nie zmieniany,
- Z — nie zmieniany,
- V — nie zmieniany,
- C — nie zmieniany.

Dozwolone tryby adresowania:

bezpośredni tryb adresowania rejestru adresowego

Dodatek IV**Tablice czasu wykonywania poszczególnych instrukcji MC 68000****Używane symbole:**

- B — bajt,
 W — słowo,
 L — długie słowo,
 An — rejestr adresowy,
 Dn — rejestr danych,
 ea — operand określony przez adres efektywny,
 M — operandem jest adres efektywny,
 # — operand natychmiastowy.

Czas obliczenia adresu efektywnego

Tryb adresowania		B, W	L
Dn	Tryb adresowania bezpośredniego rejestru danych	0	0
An	Tryb adresowania bezpośredniego rejestru adresowego	0	0
(An)	Tryb adresowania pośredniego rejestrem adresowym	4	8
(An)+	Tryb adresowania pośredniego rejestrem adresowym z postinkrementacją	4	8
-(An)	Tryb adresowania pośredniego rejestrem adresowym z predekrementacją	6	10
(d16,An)	Tryb adresowania pośredniego rejestrem adresowym z przesunięciem	6	12
(d8,An,Xn)*	Tryb adresowania pośredniego rejestrem adresowym z przesunięciem i indeksem	10	14
(xxx).W	Tryb adresowania absolutnego krótkiego	8	12

Tryb adresowania		B, W	L
(xxx).L	Tryb adresowania absolutnego długiego	12	16
(d8,PC)	Tryb adresowania pośredniego z licznikiem programu z przesunięciem	8	12
(d16,PC,Xn)*	Tryb adresowania pośredniego z licznikiem programu z przesunięciem i indeksem	10	14
#(dana)	Natychmiastowy	4	8

* rozmiar rejestru indeksowego nie ma wpływu na czas wykonywania instrukcji

Czas wykonywania instrukcji MOVE przesyłającej bajt lub słowo

Źródło	Przeznaczenie								
	Dn	An	(An)	(An)+	-(An)	(d16,An)	(d8,An,Xn)*	(xxx.W)	(xxx.L)
Dn	4	4	8	8	8	12	14	12	16
An	4	4	8	8	8	12	14	12	16
(An)	8	8	12	12	12	16	18	16	20
(An)+	8	8	12	12	12	16	18	16	20
-(An)	10	10	14	14	14	18	20	18	20
(d16,An)	12	12	16	16	16	20	22	20	24
(d8,An,Xn)*	14	14	18	18	18	22	24	22	26
(xxx.W)	12	12	16	16	16	20	22	20	24
(xxx.L)	16	16	20	20	20	24	26	24	28
(d16,PC)	12	12	16	16	16	20	22	20	24
(d8,PC,Xn)*	14	14	18	18	18	22	24	22	26
#dana	8	8	12	12	12	16	18	16	20

* rozmiar rejestru indeksowego nie ma wpływu na czas wykonywania instrukcji

Czas wykonywania instrukcji MOVE przesyłającej długie słowo

Źródło	Przeznaczenie								
	Dn	An	(An)	(An)+	-(An)	(d16,An)	d8,An,Xn)*	(xxx.W)	(xxx.L)
Dn	4	4	12	12	12	16	18	16	20
An	4	4	12	12	12	16	18	16	20
(An)	12	12	20	20	20	24	26	24	28
(An)+	12	12	20	20	20	24	26	24	28
-(An)	14	14	22	22	22	26	28	26	30
(d16,,An)	16	16	24	24	24	28	30	28	32
(d8,An,Xn)*	18	18	26	26	26	30	32	30	34
(xxx.W)	16	16	24	24	24	28	30	28	32
(xxx.L)	20	20	28	28	28	22	34	32	36
(d16,PC)	16	16	24	24	24	28	30	28	32
(d8,PC,Xn)*	18	18	26	26	26	30	32	30	34
#dana	12	12	20	20	20	24	26	24	28

rozmiar rejestru indeksowego nie ma wpływu na czas wykonywania instrukcji

Czas wykonywania standardowych instrukcji

Instrukcja	Rozmiar	operandy: <ea>,An	operandy: <ea>,Dn	operandy: Dn,<M>
ADD/ADDA	B, W	8+	4+	8+
	L	6+	6+	12+
AND	B, W	—	4+	8+
	L	—	6+	12+
CMP/CMPA	B, W	6+	4+	—
	L	6+	6+	—
DIVS	—	—	158+	—

Instrukcja	Rozmiar	operandy: <ea>,An	operandy: <ea>,Dn	operandy: Dn,<M>
DIVU	—	—	140+	—
EOR	B, W	—	4	8+
	L	—	8	12+
MULS	—	—	70+	—
MULU	—	—	70+	—
OR	B, W	—	4+	8+
	L	—	6+	12+
SUB	B, W	8+	4+	8+
	L	6+	6+	12+

+ należy dodać czas obliczenia adresu efektywnego

* tylko słowo lub długie słowo

Czas wykonywania instrukcji natychmiastowych

Instrukcja	Rozmiar	operandy: #,Dn	operandy: #,An	operandy: #,M
ADDI	B, W	8	—	12+
	L	16	—	20+
ADDQ	B, W	4	4*	8+
	L	8	8	12+
ANDI	B, W	8	—	12+
	L	14	—	20+
CMPI	B, W	8	—	8+
	L	14	—	12+
EORI	B, W	8	—	12+
	L	16	—	20+
MOVEQ	L	4	—	—

Instrukcja	Rozmiar	operandy: #,Dn	operandy: #,An	operandy: #,M
ORI	B, W	8	—	12+
	L	16	—	20+
SUBI	B, W	8	—	12+
	L	16	—	20+
SUBQ	B, W	4	8*	8+
	L	8	8	12+

+ należy dodać czas obliczenia adresu efektywnego

* tylko słowo lub długie słowo

Czas wykonywania instrukcji o pojedynczym operandzie

Instrukcja	Rozmiar	Rejestr	Pamięć
CLR	B, W	4	8+
	L	6	12+
NBCD	B	6	8+
NEG	B, W	4	8+
	L	6	12+
NEGX	B, W	4	8+
	L	6	12+
NOT	B, S	4	8+
	L	6	12+
Sxx	B, Fałsz	4	8+
	B, Prawda	6	8+
TAS	B	4	14+
TST	B, W	4	4+
	L	4	4+

+ należy dodać czas obliczenia adresu efektywnego określonego odpowiednim trybem adresowania

Czas wykonywania instrukcji obracających lub przesuwających bity

Instrukcja	Rozmiar	Rejestr	Pamięć
ASR, ASL	B, W	$6 + 2n$	8+
	L	$8 + 2n$	—
LSR, LSL	B, W	$6 + 2n$	8+
	L	$8 + 2n$	—
ROR, ROL	B, W	$6 + 2n$	8+
	L	$8 + 2n$	—
ROXR, ROXL	B, W	$6 + 2n$	8+
	L	$8 + 2n$	—

+ należy dodać czas obliczenia adresu efektywnego określonego odpowiednim trybem adresowania
 n – przesunięcie

Czas wykonywania instrukcji manipulujących bitami

Instrukcja	Rozmiar	Dynamiczny		Statyczny	
		Rejestr	Pamięć	Rejestr	Pamięć
BCHG	B, W	—	8+	—	12+
	L	8*	—	12*	—
BCLR	B, W	—	8+	—	12+
	L	10*	—	14*	—
BSET	B, W	—	8+	—	12+
	L	8*	—	12*	—
BTST	B, W	—	4+	—	8+
	L	6*	—10	—	

+ należy dodać czas obliczenia adresu efektywnego określonego odpowiednim trybem adresowania dla maksymalnej wartości; tylko tryb adresowania danych

Czas wykonywania instrukcji warunkowych i skoków

Instrukcja	Rozmiar	Skok wykonany	Skok niewykonany
BCC	B	10	8
	W	10	12
BRA	B	10	—
	W	10	—
BSR	B	18	—
	W	18	—
DBxx	xx prawda	—	12
	xx niespełniony, licznik niekontynuowany	10	—
	xx niespełniony, licznik kontynuowany	—	14

Czas wykonywania instrukcji JMP, JSR, LEA, PEA, i MOVEM

Instrukcja	Size	(An)	(An)+	-(An)	(d16,An)	(d8,An,Xn)
JMP	—	8	—	—	10	14
JSR	—	16	—	—	18	22
LEA	—	4	—	—	8	12
PEA	—	12	—	—	16	20
MOVEM P→R	W	12+4n	12+4n	—	16+4n	18+4n
	L	12+8n	12+8n	—	16+8n	18+8n
MOVEM R→P	W	8+4n	—	8+4n	12+4n	4+4n
	L	8+8n	—	8+8n	12+8n	4+8n

Instrukcja	Size	(xxx).W	(xxx).L	(d16,PC)	(d8,PC,Xn)*
JMP	—	10	12	10	14
JSR	—	18	20	18	22
LEA	—	8	12	8	12
PEA	—	16	20	16	20
MOVEM P→R	W	16+4n	20+4n	16+4n	18+4n
	L	16+8n	20+8n	16+8n	18+8n
MOVEM R→P	W	12+4n	16+4n	—	—
	L	12+8n	16+8n	—	—

n – ilość rejestrów do przesłania

* rozmiar rejestru indeksowego nie ma wpływu na czas wykonywania instrukcji

Czas wykonywania instrukcji wielokrotnej precyzji

Instrukcja	Rozmiar	operandy: Dn,Dn	operandy: M,M
ADDX	B, W	4	18
	L	8	30
CMPM	B, W	—	12
	L	—	20
SUBX	B, W	4	28
	L	8	30
ABCD	B	6	18
SBCD	B	6	18

Czas wykonywania instrukcji MOVEP

Instrukcja	Rozmiar	Rejestr → Pamięć	Rejestr → Pamięć
MOVEP	W	16	16
	L	24	24

Czas wykonywania pozostałych instrukcji

Instrukcja	Rozmiar	Rejestr	Pamięć
ANDI to CCR	B	20	—
ANDI to SR	W	20	—
CHK (No Trap)	—	10+	—
EORI to CCR	B	20	—
EORI to SR	W	20	—
ORI to CCR	B	20	—
ORI to SR	W	20	—
MOVE from SR	—	6	8+
MOVE to CCR	—	12	12+
MOVE to SR	—	6	12+
EXG	—	6	—
EXT	W	4	—
	W	4	—
LINK	—	16	—
MOVE from USP	—	4	—
MOVE to USP	—	4	—
NOP	—	4	—
RESET	—	132	—
RTE	—	20	—
RTR	—	20	—
RTS	—	16	—
STOP	—	4	—
SWAP	—	4	—
TRAPV	—	4	—
UNLK	—	12	—

+ należy dodać czas obliczenia adresu efektywnego określonego trybem adresowania,

Dodatek V

Opis programu ASM ONE

W związku z tym, że na naszym rynku literatury komputerowej praktycznie nie ma opisu żadnego porządnego asemblera (oprócz MASTER SEKI), postanowiłem zamieścić tutaj opis bardzo dobrego i ogólnie dostępnego asemblera ASM ONE. Opis ten jest co prawda bardzo ogólny, niemniej powinien pomóc początkującym w obsłudze tego dość zawiłego programu. Jednak czytanie tego opisu nic nie da, jeśli nie będziesz eksperymentował z programem.

Pracę z asemblerem rozpoczyna się od udzielenia odpowiedzi na pytania dotyczące typu pamięci i wielkości obszaru roboczego, który chcemy zapewnić dla programu. Pamięć możemy przydzielić w obszarze CHIP, FAST lub pod adresem absolutnym. Należy pamiętać, że jeśli chcemy pisać program z copperlistą, to musi się ona znajdować w pamięci typu CHIP. Ilość pamięci jaką przydzielimy, zależy tylko od objętości projektu. Na początek 100 kB powinno wystarczyć.

Po zarezerwowaniu obszaru roboczego, można rozpocząć pracę. Wszystkie funkcje są dostępne z menu (prawy klawisz myszy) lub z klawiatury (poprzez wpisanie komendy lub poprzez wcisnięcie wyróżnionego klawisza wraz z klawiszem AMIGA (A)).

Przejdźcie do edytora (w nim wpisujemy tekst źródłowy programu) następuje po naciśnięciu klawisza ESC. Ten sam klawisz służy do wyjścia z edytora, monitora, czy debuggera.

Jeśli nie jesteśmy w edytorze, monitorze lub debuggerze, to znajdujemy się na ekranie roboczym, na którym możemy wybierać funkcje z trzech menu: PROJECT, ASSEMBLER, COMMAND, lub też wpisywać polecenia z klawiatury. Litery w nawiasach są komendami, które można wpisać na ekranie roboczym i które powodują wywołanie danej opcji.

Uwaga: jeśli chcesz natychmiast uruchomić napisany program, to musisz najpierw go zasemblować opcją ASSEMBLE, a potem uruchomić za pomocą komendy Jump lub Go.

a. Menu Project

Zap Source (ZS) — usuwa tekst źródłowy z edytora. Tekst nie jest natychmiast usuwany z pamięci, a jedynie edytor jest poinformowany o zerowej długości pliku. Tekst można odzyskać funkcją "Old" (O).

Old (O) — przywraca tekst źródłowy skasowany funkcją "Zap Source". Komenda ta podaje także wielkości obszarów roboczych.

Read — zawiera zestaw trzech komend służących do wczytywania plików do pamięci.

Read Source (R) — wczytuje tekst źródłowy do edytora. Poprzedni tekst źródłowy zostaje usunięty z pamięci.

Read Binary (RB) — doczytuje dane pod określony adres. Podajemy dane początku i końca obszaru przeznaczonego na wczytywanie. Pomijając adres końca (nie wpisując i od razu

naciskając klawisz ENTER), powodujemy doczytanie całego pliku do pamięci. Należy jednak z tą opcją uważać i upewnić się, że obszar przeznaczony na doczytywanie jest wolny.

Read Object (RO) — wczytuje plik wykonywalny (np. program) do pamięci.

Po zakończeniu operacji, podawany jest adres początku doczytanego pliku.

Write — zawiera zestaw czterech komend służących do wczytywania plików do pamięci.

Write Source (W) — zapisuje tekst źródłowy z edytora na dysk.

Write Binary (WB) — zapisuje dane spod określonego adresu na dysk. Podajemy dane początku i końca obszaru przeznaczonego do zapisu.

Write Object (WO) — nagrywa zasemblowany kod jako plik wykonywalny. Jednak niektóre wersje ASM ONE mają tę opcję uszkodzoną.

Write Link (WL) — nagrywa na dysk zasemblowany program jako plik dołączalny.

Insert (I) — umożliwia dołączenie pliku źródłowego do tekstu źródłowego znajdującego się w edytorze. Plik dołączany jest na pozycji kursora w edytorze.

Update (U) — umożliwia zapisanie tekstu źródłowego z edytora pod starą nazwą. Poprzedni plik zostanie usunięty z dysku.

Zap File (ZF) — usuwa z dysku wybrany plik.

Preferences — umożliwia ustawienie preferencji programu. Mamy tutaj:

Rescue — po powrocie z procedury, zostanie ustawiona systemowa copperlista.

Level 7 — umożliwia przerwanie działania każdego uruchomionego programu za pomocą specjalnego przełącznika, wywołującego przerwanie niemaskowalne (7 poziom). Niestety przełącznik taki, jak i jego schemat, są praktycznie niedostępne.

NumLock — po włączeniu tej funkcji, klawiatura numeryczna działa jako panel sterujący kursorem w edytorze.

ReqLibrary — ASM ONE będzie używał w celu komunikacji z użytkownikiem biblioteki REQ.LIBRARY. Będzie się z nim komunikować za pomocą requesterów, okienek, itp.

Interlace — ekran roboczy zostanie uruchomiony w trybie interlace (512 linii w pionie).

1 Bitplane — uruchamia ekran roboczy w 2 kolorach (standardowo są cztery). Daje to większą pamięć do dyspozycji.

Source.S — program będzie doczytywał tylko teksty źródłowe z dopełnieniem .S.

CloseWB — WORKBENCH po uruchomieniu programu zostanie zamknięty.

WritePref (WP) — zapisuje preferencje na dysku. Będą one automatycznie doczytywane przy ponownym uruchomieniu programu.

About — informacje o autorze, wersji programu, itp..

Exit (!) — wyjście z programu lub jego powtórne uruchomienie w celu zmiany obszaru roboczego.

b. Menu Assembler

Assemble — zawiera opcje asemblacji i optymalizacji oraz preferencje dla procedury asemblującej.

Assemble (AMIGA-A lub A) — dokonuje asemblacji tekstu źródłowego zawartego w edytorze.

Asemblacja oraz optymalizacja mogą zostać przerwane za pomocą kombinacji CTRL-C.

Optimize (AMIGA-O) — dokonuje optymalizacji tekstu źródłowego oraz asemblacji.

ListFile — opcja ta powoduje, że przy asemblacji pokazywany jest tekst źródłowy oraz kody na jakie został zasemblowany.

Pagging — gdy włączona jest opcja ListFile, to listing zostaje podzielony na strony.

HaltPage — po każdej stronie wylistowanej podczas asemblacji, następuje zatrzymanie listowania i oczekiwanie na klawisz.

AllErrors — podczas asemblacji będą pokazywane wszystkie błędy.

Debug — do zasemlowanego pliku uruchamialnego dołączany jest zbiór debug umożliwiający później łatwiejsze przeglądanie zbiorów za pomocą monitora lub debuggera.

Label: — gdy opcja ta jest włączona, to wszystkie etykiety muszą być zakończone dwukropkiem.

UCase=LCase — jeśli opcja ta jest wyłączona, to w etykietach rozróżniane są duże i małe litery. Wtedy np. etykiety Start i START będą traktowane jako dwie różne.

Editor (ESC lub AMIGA-E) — przejście do edytora.

Debugger (AMIGA-D) — przejście do debuggera. Najpierw jednak następuje asemblacja tekstu źródłowego.

Monitor (AMIGA-M) — przejście do monitora.

LineNumbers — po włączeniu tej opcji, każda linia w edytorze jest ponumerowana.

AutoIndent — gdy pracujemy w edytorze to po naciśnięciu klawisza ENTER następuje przejście do następnej linii z jednoczesną tabulacją taką samą jak w linii poprzedniej.

c. Menu Command

Editor — funkcje dotyczące edytora.

Top (T) — przejście na początek tekstu źródłowego.

Bottom (B) — przejście na koniec tekstu źródłowego.

Search (L) — poszukuje danego ciągu znaków w tekście źródłowym.

Zap Line (ZL) — kasuje linię w tekście źródłowym zawartym w edytorze, począwszy od aktualnej pozycji kursora. Podanie przy komendzie jakąś liczbę, to zostanie ona potraktowana jako ilość linii do skasowania.

PrintLine (P) — drukuje żadaną ilość linii na drukarce.

Memory — operacje na pamięci.

Edit (M) — edycja pamięci pod podanym adresem jako bajty (w zapisie heksadecymalnym).

DisAssem (D) — disasemblacja pamięci od podanego adresu.

HexDump (H) — przeglądanie pamięci od podanego adresu jako wartości w kodzie szesnastkowym.

Ascii (N) — przeglądanie pamięci od podanego adresu jako kodów ASCII.

DisLine (@D) — disasemblacja pamięci w podanym zakresie (jako listing).

Assemble (@A) — bezpośrednia asemblacja do pamięci (pominięcie tekstu źródłowego).

HexLine (@H) — przeglądanie pamięci jako wartości w kodzie szesnastkowym w podanym zakresie.

AsciiLine (@N) — przeglądanie pamięci jako kodów ASCII w podanym zakresie.

Search (S) — wyszukiwanie w pamięci pewnych wartości (podajemy obszar pamięci do przeszukania i szukane wartości).

Fill (F) — wypełnienie podanego obszaru pamięci daną wartością.

Copy (C) — skopiowanie podanego obszaru pamięci w inne miejsce.

Compare (Q) — porównuje zawartość jednego obszaru pamięci z innym i podaje adresy tych komórek, w których wystąpiła niezgodność.

Insert — pozwala na dołączenie danych znajdujących się w pamięci do kodu źródłowego.

DisAssem (ID) — dane z określonego obszaru pamięci zostają poddane diasemblacji i dołączone do tekstu źródłowego.

- HexDump (IH) — dane z określonego obszaru pamięci zostają dołączone do tekstu źródłowego jako wartości szesnastkowe.
- Ascii (IN) — dane z określonego obszaru pamięci zostaną dołączone jako tekst ASCII.
- Assemble — zawiera funkcje dotyczące asemblacji.
- Assemble (A) — asemblacja tekstu źródłowego znajdującego się w edytorze.
- Memory (@A) — bezpośrednia asemblacja do pamięci (pominięcie tekstu źródłowego).
- Optimize (AO) — optymalizacja i asemblacja tekstu źródłowego znajdującego się w edytorze.
- Debug (AD) — przejście do debuggera.
- Symbols (=S) — lista użytych symboli (etykiet) w zasemblowanym kodzie źródłowym.
- Monitor — funkcje dotyczące uruchamiania programu (zasemblowanego tekstu źródłowego).
- Jump (J) — skok do programu (działa jak JSR). Po zakończeniu wykonywania programu, następuje powrót do ekranu roboczego.
- Go (G) — skok do programu (działa jak JMP). Po zakończeniu wykonywania programu powrót nastąpi na wartość odłożoną na stosie, czyli na ogół w zupełnie przypadkowe miejsce. Po napotkaniu nieistniejącej instrukcji, generowany zostaje stan wyjątkowy i następuje powrót do ekranu roboczego.
- Step (K) — wykonywanie programu krokowo, po jednej instrukcji.
- Status (X) — podaje zawartość rejestrów procesora (jeśli podamy nazwę rejestru, to możemy zmienić jego zawartość).
- Zap BPS (ZB) — usuwa pułapki z programu.
- Disk — funkcje dotyczące obsługi stacji dysków.
- Read Sector (RS) — wczytanie do pamięci podanych sektorów.
- Read Track (RT) — wczytanie do pamięci podanych ścieżek.
- Write Sector (WS) — zapisanie podanego fragmentu pamięci na dysku, do określonych sektorów.
- Write Track (WT) — zapisanie podanego fragmentu pamięci na dysku, do określonych ścieżek.
- CalacCheck (CC) — Obliczenie sumy kontrolnej bootblocku i zapisanie jej na dysku. Można podać także numer stacji dysków (np. "CC 1" — stacja DF1:).
- Extern (E) — wykonanie wszystkich komend EXTERN znajdujących się w tekście źródłowym.
- Output (>) — skierowanie wyjścia danych na inne urządzenie (np. na drukarkę, czy na dysk).
- Calculate (?) — oblicza wartość podanego wyrażenia. Zamiast wartości liczbowych, możemy używać symboli lub etykiet zawartych w kodzie wynikowym.

Po przejściu do edytora (np. klawiszem ESC) zostaje udostępnione nowe menu.

d. Menu Edit Funct

Block — dotyczy operacji na bloku tekstu wyciętym z tekstu źródłowego.

Mark (AMIGA-B) — zaznaczenie początku bloku (w miejscu aktualnej pozycji kursora).

Copy (AMIGA-C) — skopiowanie zaznaczonego bloku do bufora.

Cut (AMIGA-X) — wycięcie zaznaczonego bloku z tekstu źródłowego i skopiowanie go do bufora.

Insert (AMIGA-I) — dołączenie bloku znajdującego się w buforze do tekstu źródłowego na pozycji kursora.

Fill (AMIGA-F) — jak Insert.

UnMark (AMIGA-U) — anuluje zaznaczone miejsce początku bloku.

LowerCase (AMIGA-L) — wszystkie litery w bloku zostaną zamienione na małe.

UpperCase (AMIGA-L) — wszystkie litery w bloku zostaną zamienione na duże.

Rotate (AMIGA-Y) — odwrócenie kolejności linii w zaznaczonym bloku.

Registers (AMIGA-K) — podaje rejestry używane w zaznaczonym bloku.

VerFill (AMIGA-N) — wstawienie zawartości bufora do tekstu źródłowego i przejście kursora do następnej linii.

Search — służy do wyszukiwania ciągu znaków w tekście źródłowym.

Search (AMIGA-S) — wpisanie ciągu znaków do poszukiwania i odszukanie pierwszego ciągu poniżej aktualnej pozycji kursora.

Forward (AMIGA-S) — wyszukiwanie następnych ciągów.

Replace — służy do zastępowania jednego ciągu znaków, innym.

Replace (AMIGA-R) — wpisanie ciągów do wyszukania i do zamiany oraz wyszukanie pierwszego ciągu, który może być zmniejszony. Pojawią się wtedy na górze ekranu opcje: Y/N/L/G.

Y — zamienia ciąg i wyszukuje następny.

N — pomija ciąg i wyszukuje następny.

L — zamienia ciąg i wyszukuje następny.

G — zamienia wszystkie napotkane ciągi w tekście źródłowym.

Forward (AMIGA-R) — wyszukuje i zamienia ciąg znaków.

Del Line (AMIGA-D) — skasowanie linii i skopiowanie jej do bufora.

Marks — oznaczenie stałych pozycji w tekście, do których można się przenieść w każdej chwili.

Mark 1 — zaznaczenie pierwszej pozycji w tekście (przyjmuje ona aktualną pozycję kursora).

Mark 2 — zaznaczenie drugiej pozycji.

Mark 3 — zaznaczenie trzeciej pozycji.

Jump 1 — skok do pierwszej pozycji.

Jump 2 — skok do drugiej pozycji.

Jump 3 — skok do trzeciej pozycji.

Jump ;; (AMIGA-J) — skok do miejsca oznaczonego dwoma średnikami.

Jump Line (AMIGA-J) — wykonuje skok do linii o podanym numerze.

Move — zawiera funkcje przesuwania kursora.

MakeMacro (AMIGA-) — funkcja ta umożliwia stworzenie makrobloku.

DoMacro (AMIGA-M) — wstawia makroblok na pozycji kursora.

Po naciśnięciu kombinacji AMIGA-M przechodzimy do monitora i mamy tam nowe menu.

e. Menu Mon Funct

DisAssem (AMIGA-D) — przejście do trybu disasemblacji.

HexDump (AMIGA-H) — wszystkie dane będą wyświetlane szesnastkowo.

Ascii (AMIGA-N) — wszystkie dane będą pokazywane jako znaki ASCII.

Jump Addr (AMIGA-J) — ustawienie początku wyświetlanego obszaru pamięci.

Last Addr (AMIGA-L) — ustawienie początku wyświetlanego na ostatnim adresie.

Mark i Jump — działają identycznie jak w edytorze.

QuickJump (AMIGA-Q) — pobiera adres z danych i umożliwia skok pod ten adres, jeśli potwierdzimy to klawiszem ENTER.

Only Ascii — Jeżeli opcja ta będzie uaktywniona, to wyświetlane będą tylko kody ASCII z zakresu od 32 do 127.

Ostatnim, dostępnym po wejściu do debuggera, menu jest Debug Funct.

f. Menu Debug Funct

Step One (↓) — wykonywanie rozkazów instrukcja po instrukcji, lecz wszystkie instrukcje skoków wykonywane są w całości.

Enter (→) — wejście do procedury, do której aktualnie wykonywany jest aktualnie skok (BSR lub JSR).

Run (AMIGA-R) — uruchomienie programu. Powrót do debuggera nastąpi, gdy program napotka pułapkę lub gdy program zakończy swe działanie.

Step n (AMIGA-S) — wykonanie n instrukcji programu i zatrzymanie go.

Edit Regs (AMIGA-X) — umożliwia zmianę zawartości danego rejestru.

AddWatch (AMIGA-A) — umożliwia sprawdzenie zmian w pamięci, w czasie wykonywania programu. Pokazuje dany fragment pamięci jako wartości liczbowe lub kody ASCII.

DelWatch — kasuje wcześniej ustawione adresy do oglądania pamięci.

ZapWatch (AMIGA-Z) — usuwa wszystkie adresy dodane funkcją AddWatch.

Jump (AMIGA-J) — zmiana adresu do analizy działania programu.

Jump Mark (AMIGA-J) — ustalenie nowego adresu do analizy działania programu. Dokonujemy tego przesuając kursorem na daną linię programu (lub odpowiedni adres).

B.P. Addr (AMIGA-B) — umożliwia zastawienie pułapki pod podanym adresem. W miejscu, w którym została zastawiona pułapka, działanie programu zostanie przerwane i nastąpi powrót do debuggera.

B.P. Mark (AMIGA-B) — ustawienie pułapki poprzez ustawienie kursora w żądanym miejscu i naciśnięciu klawisza ENTER.

Zap all B.P. (AMIGA-Z) — usuwa wszystkie zastawione wcześniej pułapki.

Funkcje blokowe:

AMIGA-B	lub	CTRL-B	— zaznaczenie bloku
AMIGA-C	lub	CTRL-C	— kopiowanie bloku
AMIGA-X	lub	CTRL-X	— wycięcie bloku
AMIGA-I	lub	CTRL-I	— wstawienie bloku
AMIGA-F	lub	CTRL-F	— wstawienie bloku
AMIGA-N	lub	CTRL-N	— wstawienie bloku
AMIGA-U	lub	CTRL-U	— kasowanie zaznaczonego bloku
AMIGA-L	lub	CTRL-L	— zamiana na małe litery
AMIGA-L	lub	CTRL-L	— zamiana na duże litery
AMIGA-Y	lub	CTRL-Y	— rotacja bloku
AMIGA-K	lub	CTRL-K	— wykaz używanych rejestrów w zaznaczonym bloku
AMIGA-W	lub	CTRL-W	— zapis bloku na dysk

Szukanie i zamiana:

AMIGA-S	lub	CTRL-S	— szukanie
AMIGA-S	lub	CTRL-S	— kontynuacja szukania
AMIGA-R	lub	CTRL-R	— szukanie i zamiana
AMIGA-R	lub	CTRL-R	— kontynuacja szukania i zamiany

Rozkazy skoku:

AMIGA-!	lub	CTRL-!	— zaznaczenie 1 punktu skoku
AMIGA-@	lub	CTRL-@	— zaznaczenie 2 punktu skoku
AMIGA-#	lub	CTRL-#	— zaznaczenie 3 punktu skoku
AMIGA-1	lub	CTRL-1	— skok do 1 punktu
AMIGA-2	lub	CTRL-2	— skok do 2 punktu
AMIGA-3	lub	CTRL-3	— skok do 3 punktu
AMIGA-J	lub	CTRL-J	— skok do wiersza z >;<
AMIGA-J	lub	CTRL-J	— skok do wiersza...

Ruch kursora:

SHIFT-↑	—	strona w górę	
SHIFT-↓	—	strona w dół	
SHIFT-←	—	początek wiersza	
SHIFT-→	—	koniec wiersza	
ALT-←	—	słowo w lewo	
ALT-→	—	słowo w prawo	
AMIGA-A	lub	CTRL-A	— 100 wierszy do góry
AMIGA-Z	lub	CTRL-Z	— 100 wierszy na dół
AMIGA-T	lub	CTRL-T	— początek tekstu
AMIGA-T	lub	CTRL-T	— koniec tekstu

Szybkie kasowanie:

CTRL-DEL	— kasowanie do początku wiersza
CTRL-BACK	— kasowanie od początku wiersza
AMIGA-D lub CTRL-D	— kasowanie całego wiersza

Pozostałe:

AMIGA-M	lub	CTRL-M	— wykonanie makrobloku
AMIGA-,	lub	CTRL-,	— definicja makrobloku
AMIGA-G	lub	CTRL-G	— zapis słowa w buforze

Na następnych stronach przedstawione jest krótkie zestawienie wszystkich omówionych opcji:

- [...] — parametr obowiązkowy
- (...) — parametr nieobowiązkowy
- (.) — rozmiar parametru (.B., .W., .L)
- .L] — max. długość parametru

Project:

- ZS — kasowanie tekstu źródłowego
- O — odtworzenie skasowanego tekstu źródłowego oraz rozmieszczenie danych w pamięci
- R — wczytanie tekstu źródłowego
- RB — wczytanie danych
- RO — wczytanie programu
- W — zapis tekstu źródłowego
- WB — zapis danych
- WL — zapis programu linkowego

- I — wczytanie tekstu źródłowego w miejsce kursora
- U — zapis tekstu źródłowego pod tą samą nazwą
- Z — kasowanie pliku na dysku
- ZI — kasowanie danych w pamięci
- WP — zapis ASM-One.pref
- =M — zmiana wielkości obszaru roboczego
- ! — wyjście z ASM ONE

Editor:

- TF [nr wiersza] — ustawienie kursora i wydruk wiersza
- LF [tekst] — szukanie tekstu poniżej kursora
- ZLF [ilość wierszy] — kasowanie wierszy poniżej kursora
- PF [ilość wierszy] — wydruk wierszy poniżej kursora
- BF — ustawienie kursora w ostatnim wierszu

Memory:

- M (.) (adres) — monitor
- D (adres) — disasemblacja pamięci
- H (.) (adres) — edycja pamięci w Hex
- N (adres) — edycja pamięci w ASCII
- @D (adres) — liniowa disasemblacja pamięci
- @A (adres) — liniowa asemblacja pamięci
- @H (.) (adres) — edycja liniowa pamięci w Hex
- @N (adres) — edycja liniowa pamięci w ASCII
- S (.) — szukanie w pamięci
- F (.) — wypełnianie pamięci
- C (.) — kopiowanie pamięci
- Q — porównywanie pamięci

Wstawianie:

- ID — wstawianie pamięci w miejsce kursora
- IH (.) — wstawienie pamięci w Hex w miejsce kursora
- IH — wstawienie pamięci w ASCII w miejsce kursora

Asembler:

- A — asemblacja tekstu źródłowego
- @A (adres) — liniowa asemblacja pamięci
- AO — asemblacja z optymalizacją
- AD — asemblacja i wejście do debuggera
- =S — tablica użytych etykiet

Monitor:

- J (adres) — wykonanie instrukcji JSR adres
- G (adres) — wykonanie instrukcji JMP adres
- K (krok) — praca krokowa
- X (rejstr) — zawartość rejestrów
- ZB — kasowanie wszystkich pułapek

Dysk:

RS (drive)	— czytanie sektorów
RT (drive)	— czytanie tracków
WS (drive)	— zapis sektorów
WT (drive)	— zapis tracków
CC (drive)	— obliczanie sumy BootBlocku
E	— załadowanie wszystkich Externów
V (drive)	— katalog dysku
>	— otworenie lub zamknięcie pliku OUTPUT

Pozostałe:

?	— kalkulator
=	— rozmieszczenie danych w pamięci
CS	— tworzenie tablicy sinusów w pamięci
CSS	— tworzenie tablicy sinusów w programie źródłowym

Rozkazy kontroli asemblacji

SECTION — początek programu

(etykieta) SECTION [nazwa], [typ]

CODE — wytworzenie relokowalnej sekcji kodu

DATA — wytworzenie danych

BSS — wytworzenie hunku samych zer

_C — umiejscowienie w Chip-RAM

_F — umiejscowienie w Fast-RAM

_P — umiejscowienie pamięci dowolnego typu

RORG — względny adres startu programu

(etykieta) RORG [.L]

ORG — absolutny adres startu programu

(etykieta) ORG [.L]

LOAD — absolutny adres ładowania

(etykieta) LOAD [.L]

OFFSET — definicja Offset

(etykieta) OFFSET [.L]

ENDOFF — koniec tabeli Offset

(etykieta) ENDOFF

END — koniec programu

(etykieta) END

BASEREG — ustawienie bazy rejestru

(etykieta) BASEREG [adres], [An]

Rozkazy definicji danych:

- DC** — definicja stałej
(etykieta) DC [.] [dana], (dana)
- DCB** — definicja bloku stałych
(etykieta) DCB [.] [.L], [.W]
- DS** — definicja wolnej przestrzeni
(etykieta) DS [.] [.L]
- BLK** — blok
(etykieta) BLK [.] [.L], [.W]
- DR** — definicja względnego słowa
(etykieta) DR [.] [.W]

Rozkazy definicji symboli:

- EQU** — przypisanie stałej za słowo
[etykieta] EQU [.W]
- SET** — przypisanie za słowo
[etykieta] SET [.W]
- EQUR** — przypisanie za rejestr
[etykieta] EQUR [Rn]
- REG** — przypisanie za listę rejestrów
[etykieta] REG [lista rejestrów]
- RS** — przypisanie względnej za słowo
(etykieta) RS [.] [.L]
- RSRESET** — wstawienie z powrotem liczby zamiast względnej Offset
(etykieta) RSRESET
- RSSET** — wstawienie liczby zamiast względnej Offset
(etykieta) RSSET [.L]

Rozkazy macro:

- MACRO** — rozpoczęcie definicji macro (etykieta) MACRO
- NARG** — symbol specjalny
- ENDM** — zakończenie definicji macro
- MEXIT** — opuszczenie macro
- CMEXIT** — opuszczenie macro jeżeli osiągnięto ponownie
CMEXIT [.L]
- REPT** — rozpoczęcie powtarzalnego kodu
REPT [.L]

ENDR — zakończenie powtarzalnego kodu

Warunki asemblacji:

CNOP — warunek NOP do przekazania w przyległym .W lub .L

(etykieta) CNOP [.L1], [.L2]

EVEN — wymuszenie kontynuacji od adresu parzystego

(etykieta) EVEN

ODD — wymuszenie nieparzystego adresu

(etykieta) ODD

IFEQ — asemblacja jeżeli =0

IFNE — asemblacja jeżeli <>0

IFGT — asemblacja jeżeli >0

IFGE — asemblacja jeżeli >=0

IFLT — asemblacja jeżeli <0

IFLE — asemblacja jeżeli <=0

IF — asemblacja jeżeli logiczna prawda

IF [warunek]

IFC — asemblacja jeżeli są identyczne

IFC [etykieta], [etykieta]

IFNC — asemblacja jeżeli nie są identyczne

IFNC [etykieta], [etykieta]

IFD — asemblacja jeżeli jest zdefiniowana etykieta

IFD [etykieta]

IFND — asemblacja jeżeli nie jest zdefiniowana etykieta

IFND [etykieta]

IFB — asemblacja jeżeli etykieta jest wolna

IFB [etykieta]

IFNB — asemblacja jeżeli etykieta nie jest wolna

IFNB [etykieta]

IF1 — asemblacja tylko w >Pass 1<

IF2 — asemblacja tylko w >Pass 2<

ELSE — alternatywny kod jeżeli nie jest spełniony warunek

ENDC — zakończenie bloku IF

Kontrola edycji:

PAGE — rozpoczęcie nowej strony

NOPAGE — łączenie podzielonych stron

LIST — włączenie listingu

NOLIST — zakończenie listingu

LLEN	— wstawienie wiersza (60 – 132)
	LLEN [.L]
PLEN	— wstawienie strony (20 – 100)
	PLEN [.L]
SPC	— wytworzenie pustej przestrzeni
	SPC [.L]
TTL	— wstawienie tytułu programu
	TTL [tekst]
FAIL	— generacja błędu asemblacji
PRINT	— wydruk tekstu na ekranie
	PRINTT [tekst], (tekst)
PRINTV	— wydruk wyrazu na ekranie
	PRINTV [wyraz], (wyraz)

Symbole Externów:

XDEF	— definicja Externa
	XDEF [etykieta], (etykieta)
XREF	— przedstawienie Externu
	XREF [etykieta], (etykieta)

Pozostałe:

JUMPPTR	— ustawienie punktu startu
	JUMPPTR [etykieta]
INCBIN	— wstawienie pliku danych
	INCBIN [nazwa], [.L]
INCLUDE	— wstawienie pliku źródłowego
	INCLUDE [nazwa]
INCDIR	— ustawienie katalogu
	INCDIR [katalog]
>EXTERN	— odczyt pliku danych
	>EXTERN [drive], [nazwa], [adres], [długość]
IDNT	— nazwa programu
	IDNT [nazwa]
AUTO	— automatyczne wykonywanie rozkazów
	AUTO [rozkaz]\(rozkaz)

Dodatek VI

Skorowidz rozkazów MC 68000

ABCD 273	DIVU 30	ORI 90
ADD 23	EOR 157	PEA 280
ADDA 66	EORI 159	RESET 269
ADDI 24	EXG 218	ROL 86
ADDQ 25	EXT 182	ROR 85
ADDX 278	JMP 179	ROXL 88
AND 52	JSR 44	ROXR 87
ANDI 53	LEA 50	RTE 142
ASL 81	LINK 281	RTS 45
ASR 80	LSL 84	SBCD 272
BCHG 167	LSR 82	STOP 269
BCLR 168	MOVE 20	SUB 26
BSET 169	MOVEA 51	SUBA 66
BSR 43	MOVEM 21	SUBI 27
BTST 43	MOVEQ 21	SUBQ 28
Bxx 45	MULS 29	SUBX 279
CHK 144	MULU 29	SWAP 68
CLR 276	NBCD 271	Sxx 270
CMP 54	NEG 90	TAS 275
CPMA 276	NEGX 274	TRAP 143
CMPI 219	NOP 166	TRAPV 143
CMPM 277	NOT 204	TST 177
DBxx 67	OR 89	UNLK 281
DIVS 31		

Kurs asemblera dla początkujących

Amiga 500-4000

Gdy minie pierwsza fascynacja komputerem, przychodzi pora na zastanowienie do czego można go wykorzystać...

Czy gry i programy użytkowe to wszystko?

Czy można samemu programować komputer?

Do tworzenia programów przeznaczone są języki programowania. Jednak wykorzystywanie ich do tworzenia animacji może zakończyć się niepowodzeniem. Obraz nie będzie się zmieniać płynnie lecz skokowo. Żeby zwiększyć szybkość działania programów trzeba tworzyć procedury w asemblerze - języku zrozumiałym dla procesora. Książka „Kurs asemblera dla początkujących”: umożliwi bezpośrednie programowanie procesora osobom, które chcą tworzyć szybkie i efektywne programy a dotychczas nie zetknęły się z takimi pojęciami jak rejestr czy lista rozkazów procesora.



ISBN 83-85701-37-0

Wydawnictwo Helion

ul. Pszczyńska 89/112, 44-100 Gliwice

☎ (32) 38-81-54 ✉ 44-100 Gliwice, skrz. poczt. 462 fax (32) 38-81-01